

DERIVATION OF SCIENTIFIC LAWS THROUGH ANALYSIS OF PROCESSES AND THE CORRESPONDING EQUATIONS

MOHAMMAD ISMAT KADIR
ROLL NO. 9405023F, SESSION 1993-94-95

A THESIS SUBMITTED TO THE DEPARTMENT OF COMPUTER
SCIENCE AND ENGINEERING IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN
ENGINEERING (COMPUTER SCIENCE AND ENGINEERING)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DHAKA, BANGLADESH
SEPTEMBER, 1999



#93587#

DERIVATION OF SCIENTIFIC LAWS THROUGH ANALYSIS OF PROCESSES AND THE CORRESPONDING EQUATIONS

Mohammad Ismat Kadir
Roll no. 9405023F, Session 1993-94-95

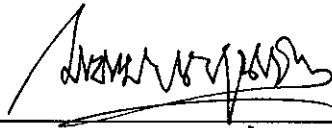
Examination held on: September 08, 1999

Approved as to style and contents by:

1. 


Dr. Chowdhury Mofizur Rahman
Assistant Professor,
Department of Computer Science and Engineering,
BUET, Dhaka, Bangladesh.

Chairman & Supervisor

2. 

Professor M. Kaykobad
Head,
Department of Computer Science and Engineering,
BUET, Dhaka, Bangladesh.

Member

3. 

Dr. Muhammad Masroor Ali
Assistant Professor,
Department of Computer Science and Engineering,
BUET, Dhaka, Bangladesh.

Member

4. 

Dr. A. B. M. Harun - ur Rashid
Assistant Professor,
Department of Electrical and Electronic Engineering,
BUET, Dhaka, Bangladesh.


Member (External)

ACKNOWLEDGEMENT

The author wishes to express his heart-felt gratitude and strong devotion to Dr. Chowdhury Mofizur Rahman, who has supervised the thesis and spent a lot of time in directing the author to the successful completion of the work. Without his constant guidance and help, the thesis would not probably come into being.

The author also expresses his gratefulness and deep sense of gratitude to Professor M. Kaykobad, Head, Computer Science and Engineering department, BUET, Dhaka, who has inspired the author in many ways to do the work.

The author wishes to thank specially Jan. M. Zytkow, who sent a large number of research papers to the author and gave valuable suggestions.



08.09.99

(Mohammad Ismat Kadir)

ABSTRACT

A system has been developed to decompose complex composite equations derived from an automated discovery system into the basic laws. AI discovery systems, like BACON, FAHRENHEIT, and ABACUS, produce algebraic equations that describe numeric data. Since physical situations in an experiment may differ in countless ways, such discovery systems have to discover equations for each of the situations individually. If there is a combination of processes in an experiment, we may have a complex and composite equation. To have a knowledge about the elementary interactions, we decompose the regularities or equations from a BACON-like discovery system into simpler expressions, each of which is associated with a simple situation or process. For example, there may be one battery and one resistor in a circuit, or there may be two or more batteries and two or more resistors in series. From the governing equation of the two circuits as obtained from a BACON-like system, the developed system will be able to generate Ohm's law and the law of equivalent resistance of a number of resistances in series. The system uses the equation from a BACON-like system as the search engine. It uses the parse tree of the equation to find a satisfactory match with the process decomposition tree. The system utilizes the dynamic database structure of PROLOG programming language to represent and store the trees. It transforms the equation into a form compatible with the physical processes they describe so that equations can be decomposed into pieces useful in model generation. We use a number of operators - more are proposed to deal with the decomposability of equations. In implementing the system, the operators are added to guide the search.

TABLE OF CONTENTS

Acknowledgement	i
Abstract	ii
Table of contents	iii
List of figures	vi
1 Introduction	1
1.1 What is an AI discovery system	1
1.2 Important features of a machine discoverer	2
1.3 Present state-of-the art of AI discovery systems	3
1.4 Automated discovery of empirical laws	4
1.5 Why discovery of elementary interactions	6
1.6 Outcome of the thesis	10
2 Present AI Discovery Systems	11
2.1 Forms of discovery	11
2.1.1 Theory-driven discovery – AM	11
2.1.2 Data – driven discovery	13
2.1.2.1 Discovering Empirical laws using BACON	13
2.1.2.1.1 A sample protocol	14
2.1.2.1.2 BACON’s representation	16
2.1.2.1.3 Production system of BACON - with special reference to BACON.3	17
2.1.2.1.4 A Summary of BACON’s discovery	20

2.1.2.1.5	BACON at work	21
2.1.2.2	Automated discovery using FAHRENHEIT	21
2.1.2.2.1	Set up experiment	21
2.1.2.2.2	Empirical space	22
2.1.2.2.3	Regularities	23
2.1.2.2.4	Exploration of numerical space	23
2.1.2.2.5	Boundary search	26
2.1.2.2.6	Generalization of empirical regularities	27
2.1.3	Clustering	28
2.2	Discovery of equations	28
2.2.1	Model-fitting and evaluation	28
2.2.2	The search space of the equation generator	31
2.2.2.1	Equation generation search	33
2.2.2.2	The search driver	33
2.2.2.3	User control over search	34
2.3	ABACUS: Integrating quantitative and qualitative discovery	36
2.3.1	The ABACUS approach to quantitative discovery	36
2.3.2	Discovering bivariate equations in ABACUS	37
2.3.2.1	Variable dependency and proportionality graph	38
2.3.2.2	Equation formation – a search for constancy	40
2.3.2.3	Domain-independent constraints	40
2.3.2.4	Recognizing the goal	41
2.3.2.5	Search	41
2.3.2.5.1	Proportionality graph search	42

2.3.2.5.2	Suspension search	44
3.	Development of the System	45
3.1	Equations and situations	45
3.1.1	Correspondence between situations and equations	45
3.2	The developed system	48
3.2.1	Representing processes and equations	49
3.2.2	Input and output	49
3.2.3	Equation transformation search	53
3.2.4	Tree matching search	54
3.2.5	Operator selection	56
3.3	The system with two case studies	56
3.3.1	Case I: Water temperature problem	58
3.3.2	Case II: Electrical circuit problem	60
4.	Discussion and Conclusion	61
4.1	Evaluation of the system	61
4.1.1	Example 1	61
4.1.2	Example 2	62
4.2	Further development	63
4.3	Conclusion	64
	References	65
	Appendix : Program Listing	69

LIST OF FIGURES

Figure 1.1	BACON discovering the ideal gas law	5
Figure 1.2	Thermal equilibrium for (a) two samples of water (b) water and a piece of ice that melts completely, (c) water and a piece of ice that melts partially	7
Figure 1.3	Moving down an inclined plane, (a) sliding, (b) rolling.	8
Figure 1.4	Two circuits: (a) one battery and one resistor, (b) two batteries and two resistors	9
Figure 2.1	Three regularities separated by boundaries	25
Figure 2.2	The intermediate state to discover some regularities and their boundaries	25
Figure 2.3	The final state of knowledge	25
Figure 2.4	Sample data and three equations with similar goodness of fit	30
Figure 2.5	The search space, generation of variables, generation of equations and model – fitting	32
Figure 2.6	Generation of new variables and equations	35
Figure 2.7	ABACUS analysis of graph example	37
Figure 2.8	Proportionality graph	39
Figure 2.9	Proportionality graph search for ideal gas law	42
Figure 2.10	Proportionality graph search path for ideal gas law	43
Figure 2.11	Partial suspension search for conservation of momentum	45
Figure 3.1	Process diagram for figure 1.2a	49
Figure 3.2	Overview of system input and process	50

Figure 3.3	Process decomposition tree1	51
Figure 3.4	Process decomposition tree2	51
Figure 3.5	Equation parse tree for Black's equation	52
Figure 3.6	Example of transformation grammar rule	53
Figure 3.7	Results of matching, (a) partial match, (b) mismatch reduction schema	55
Figure 4.1	Parse tree for ideal gas law	61
Figure 4.2	Process decomposition tree for Black's experiment	62
Figure 4.3	Parse tree for Black's law	63

Chapter 1



Introduction

1.1 What is an AI discovery system

Learning is defined as the process by which an entity acquires knowledge. It usually occurs that knowledge is already possessed by some number of other entities. These may serve as teachers. Discovery is a restricted form of learning in which one entity acquires knowledge without the help of a teacher. Sometimes it happens that there is no one in the world who has the knowledge we seek. In that case, the kind of action we have to take is called scientific discovery.

In machine learning, we typically distinguish between (1) learning as acquiring new knowledge in the form of concepts, taxonomies, regularities, and the like, and (2) learning as performance improvement and skill acquisition. Discovery applies to learning things that exist, such as the moons of Jupiter or the laws of nature. So we confront discovery with learning in the first sense, of acquisition of objective knowledge. A discoverer must be autonomous in learning something. Whether a person or a computer system, a discoverer must be equipped with its own autonomously applicable repertoire of techniques and values. Take concept learning as an example. An unbounded number of predicates can be defined by the primitives of any language, but only some make useful concepts. A teacher who understands a concept can prepare a collection of examples and describe them by suitable attributes, while a discoverer must use its own strategies for data collection and its own judgement about relevant attributes.

The notion of autonomy requires few explanations. Kepler, for example, discovered his laws from data collected by Tycho de Brahe [31]. So neither the data collection strategies nor the attributes were his own. However, his discovery was autonomous in many ways. First, Brahe's data came without a guarantee that their exploration would lead to any discovery. Kepler picked them without assurance that he would be successful. Second, Kepler used his ideas of patterns in data – generated and evaluated many patterns before he made his discoveries. Third, autonomy best applies to the whole historical process of discovery, less to a small episode. Great discoveries were

usually made possible by the contribution of many people over a long time. Uncountable observations, many attributes, and many hypotheses on planetary motion were considered before Kepler. So when autonomy is concerned, we consider the whole scientific community as a collection of discoverer.

Discoveries are not limited to 'natural' phenomena. Anything that exists but are not explored can be the subject of a discovery. For instance, we can use a discovery system to discover the computational complexity of an algorithm. As another example, consider heuristics. Heuristics are useful when they improve the behavior of a program. If cost is one of the dimensions in a domain, the least-cost heuristic is, as a rule, discovered and various other heuristics can be invented.

1.2 Important features of a machine discoverer

The number of discovery systems has been growing considerably in the last decade. The majority of work on machine discovery focussed on the reconstruction of the scientific method. Recent developments on machine discovery confirms the old thesis [18] that discovery is problem solving and that it can be carried out by computer programs. Although discoveries are abundant in everyday life, the focus on scientific discovery has drawn the main concern. Everyday concepts notoriously elude formalism, while scientific formalism has historically proved to lead to spectacular results of scientific reasoning and knowledge representation. A typical scientific phenomenon can be studied in a compact domain that is limited to a few concepts yet rich in knowledge representable in a formal way. The continued concentration on the scientific method is expected to lead to a practical success in the future. The widespread use of computers and data-acquisition equipments by scientists permits rapid installation of a successful discovery system.

Machine discovery is no different from applications of computers in modern science. Scientists develop new methods to match increasingly sophisticated problems. Then, whole scientific communities learn and apply those methods. The future of successful machine discovery systems is similar. Modern science and the automation of discovery lead to a diminishing distinction between the normative systems and established scientific practice.

1.3 Present state-of-the-art of AI discovery systems

During the last two decades, many computer programs have been constructed that simulate various aspects of scientific discovery. The earlier systems concentrated on qualitative and quantitative regularities, discovery of intrinsic concepts etc.[10] [28] [2]. Most machine discoverers get their data in a simulation, for instance BACON [11]. A still larger group of systems work on fixed, externally provided data.

Discovery in mathematics has attracted plenty of interest, but progress has been slow. Early work on discovery in the theory of numbers was originated by the AM system [13]. EURISKO [14], an extension to AM played an important role in rediscovering some useful concepts. GRAFFITI, a program developed by Fajtlowicz [31] has produced numerous interesting conjectures in graph theory.

Recently, very fast theorem-provers have been developed for geometry, allowing us to apply an automated search to propose and then prove or disprove hypotheses in large hypothesis spaces in geometry.

Discovery in databases had a tremendous progress during the last decade. It is aimed at the automated exploration of large amounts of tabular data, collected in databases. The task is limited to discovery from the prearranged data. Left out are automation of experiment and the feedback between theory formation and experimentation strategies. Results in knowledge discovery in databases (KDD) have been described in several collection of papers, typically conference proceedings [21] [25] [23] [33] and many individual papers.

The growth of discovery systems has accelerated in the last decade. Several abilities lacking in early discovery systems have been introduced, such as the ability to consider empirical context of a law [19], the ability to design experiments [9], and the ability to represent objects, states and processes [20].

The search for regularities in a multidimensional space of numerical parameters was studied in 1981, in BACON project, whose results were summarized in [12]. Although BACON was capable of rediscovering a large number of laws of physics and chemistry, it had the limitation that all the datapoints need have a single regularity. Otherwise the algorithm could not work well. Then came the FAHRENHEIT system [30], which successfully discovered complete theories of numerical

spaces, even if there had been a number of partial regularities in the dataset. In addition, FAHRENHEIT could discover knowledge about special points such as maxima and discontinuities. Several other systems, including ABACUS [2] and IDS [20] made various improvements over BACON. As for example, the ABACUS system is able to discover multiple mathematical equations for numeric data and derives explicit, logic-style description stating preconditions for the application of the equations.

Discovery systems so far stated use modules that find mathematical equations from a sequence of data. However, in science, the discovery of empirical equations is treated as an intermediate step towards more fundamental knowledge. The deeper goal is to develop theories of elementary interactions in the world, and the hidden micro-structures of things and processes. Equation transformation and their interpretation form a search space explored by GALILEO [29]. Knowledge of structure and interaction between the components of a system give the real model of situations.

1.4 Automated Discovery of Empirical Laws

Empirical scientists discover numerical regularities or equations from experimental data. They are confronted with the real-world data to make sense of it. They make hypotheses, and in order to validate them, they design and execute experiments. Scientific discovery has inspired a number of computer models. Langley *et al.* [12] present a model of data-driven scientific discovery that has been implemented as a program called BACON, named after Sir Francis Bacon, an early philosopher of science.

BACON begins with a set of variables for a problem. For example, in the study of the behavior of gases, some variables are P , the pressure on the gas, V , the volume of the gas, n , the amount of gas in moles, and T , the temperature of the gas. Physicists have long known a law, called the *ideal gas law*, that relates these variables. BACON is able to derive this law on its own. First, BACON holds the variables n and T constant, performing experiments at different pressures P_1 , P_2 , and P_3 . BACON notices that as the pressure increases, the volume V decreases. Therefore, it creates a theoretical term PV . This term is constant. BACON systematically moves on to vary the other variables. It tries an experiment with different values of T , and finds that PV changes. The two terms are linearly related with an intercept of 0, so BACON creates a new term PV/T . Finally,

BACON varies the term n and finds another linear relationship between n and PV/T . For all values of n , P , V , and T , $PV/nT = 8.32$. This is, in fact, ideal gas law. Figure 1.1 shows BACON's reasoning in a tabular format.

N	T	P	V	PV	PV/T	PV/nT
1	300	100	24.96			
1	300	200	12.48			
1	300	300	8.32	2496		
1	310			2579.2		
1	320			2662.4	8.32	
2	320				16.64	
3	320				24.96	8.32

Figure 1.1: BACON Discovering the Ideal Gas Law

BACON has been used to discover a wide variety of scientific laws, such as Kepler's third law, Ohm's law, the conservation of momentum, and Joule's law. The heuristics BACON uses to discover the ideal gas law include noting constancies, finding linear relations, and defining theoretical terms. Other heuristics allow BACON to postulate intrinsic properties of objects and to reason by analogy. For example, if BACON finds a regularity in one set of parameters, it will attempt to generate the same regularity in a similar set of parameters. Since BACON's discovery procedure is state-space search, these heuristics allow it to reach solutions while visiting only a portion of the search space. In the gas example, BACON comes up with the ideal gas law using a minimal number of experiments.

Besides BACON, FAHRENHEIT, ABACUS and some other empirical discoverers use modules that find mathematical equations in two variables from a sequence of data. FAHRENHEIT defines the problem of empirical search for knowledge in a multi-dimensional space. It autonomously explores multi-dimensional numerical spaces, accumulating knowledge with the purpose of reaching a complete theory. ABACUS integrates both the qualitative and the quantitative approach of discovering numerical laws. It formulates equations that bind subsets of observed

data, and derives explicit, logic-style descriptions stating the preconditions for the application of these equations.

A better understanding of the science of scientific discovery may lead one day to programs that display true creativity. Much more work must be done in areas of science that BACON does not model, such as determining what data to gather, choosing (or creating) instruments to measure the data, and using analogies to previously understood phenomena.

1.5 Why Discovery of Elementary Interactions

AI discovery systems, so far stated, produce algebraic equations that describe numerical data. Algebraic equations discovered by such systems give a quantitative interpretation of the different laws of nature. However, scientific knowledge exceeds a set of separate laws – it applies to an infinite variety of physical situations, and different situations are usually described by different equations. Established empirical discovery systems discover equations for each such situation individually. But the problem is, even simple physical situations can be varied in countless ways.

For example, figure 1.2 illustrates three possible versions of an experiment conducted by Joseph Black in the eighteenth century, in which two liquids are combined and their temperature is measured at equilibrium. Figure 1.3 depicts two versions of a simple mechanical experiment, and figure 1.4 shows two simple electric circuits. One can always add more wires, resistors, or transistors and can reconfigure them to produce other electric circuits. The number of modifications for each domain is clearly unbounded.

Should one apply an empirical system case by case to discover the corresponding equations for each different situation? This would result in an unbounded set of equations rather than in the parsimonious theories that we associate with modern physics and chemistry. Furthermore, if the discovered laws are limited to individual experimental situations, they will not allow for transfer of knowledge. One might try to reduce the number of laws discovered to a manageable size by discovering equations experimentally for simple situations and then deducing equations for complex situations. However, in most cases, the equations that describe different situations are either independent or mutually inconsistent since they describe different phenomena and predict different behavior.

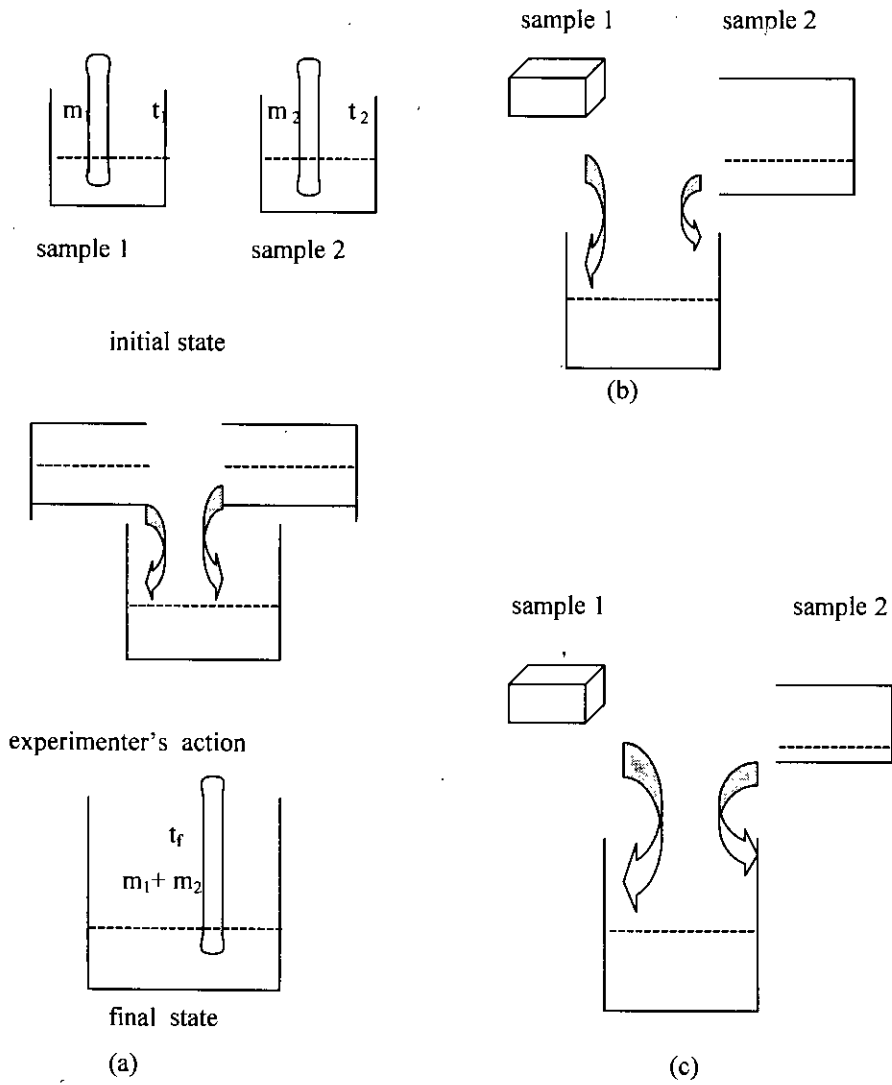
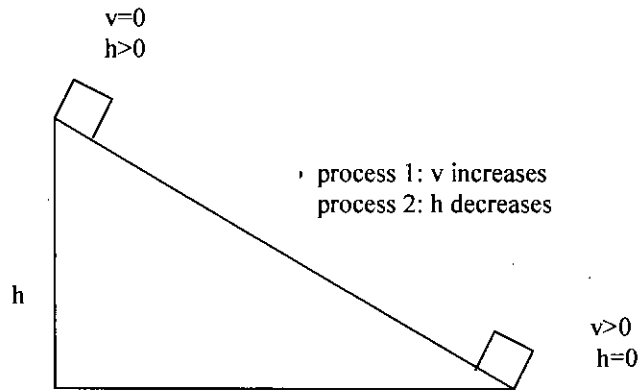
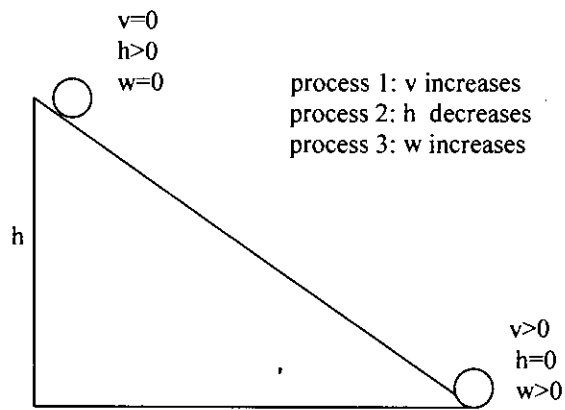


Figure 1.2 : Thermal equilibrium for (a) two samples of water, (b) water and a piece of ice that melts completely, (c) water and a piece of ice that melts partially.

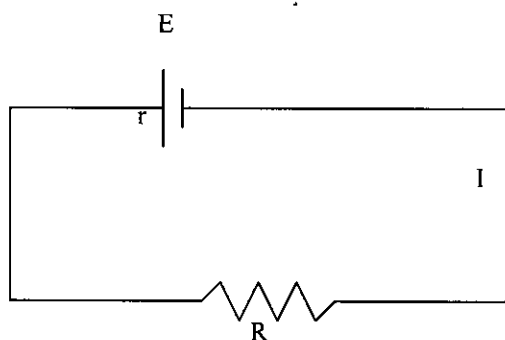


(a)

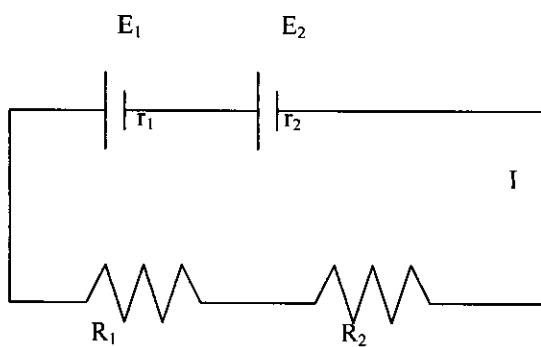


(b)

Figure 1.3 : Moving down an inclined plane: (a) sliding, (b) rolling.



(a)



(b)

Figure 1.4: Two circuits: (a) one battery and one resistor, (b) two batteries and two resistors.

BACON, FAHRENHEIT, ABACUS and similar other systems are successful in generating equations for individual physical situations, but they do not capture the way in which science deals with a multitude of physical configurations to produce simple, finite theories that can be applied to an infinite number of situations. Science deals with the complexity of physical situations by a combination of two steps. In the first step, regularities are decomposed into simpler expressions, each of which is associated with a particularly simple situation or process. In the second step, the simple expressions are recombined to form equations that model complex situations.

So we are tempted to a discovery system that transforms equations generated by any of the aforesaid system into a form compatible with the structure of physical processes they describe, so that equations can be decomposed into pieces useful in model generation.

1.6 Outcome of the thesis

The thesis is aimed at developing a computer program to decompose a complex composite equation derived from an empirical autonomous discoverer like BACON, FAHRENHEIT or ABACUS and thus to deduce the basic laws governing simple physical situations. With this end in view, we incorporate, in our system, ideas on qualitative process representation, quantitative theory and the transformation of mathematical formulas. It transforms equations generated by a BACON-like system into a form compatible with the physical processes they describe, so that equations can be decomposed into pieces useful in model generation. The outcome of our work is expected to find excellent application areas in constructing mathematical models for an integrated scientific process, given a description in terms of elementary objects and processes. An integrated scientific process is a combination of simple situations. Through an expert discovery system, like this, equations for the simple situations will be deduced from the equation of the integrated process, as obtained from a BACON-like system.

The dynamic database structure of Turbo Prolog is used to store and retrieve the integrated scientific process and the composite equation. A number of operators are implemented. The resulting program shows excellent results in some specific domain. The program is tested in cases of ideal gas law, Black's experiment, and simple electrical circuits. Thus we can treat the developed system as an integrated scientific engine to decompose complex equations into the basic laws.

Chapter 2

Present AI Discovery Systems

2.1 Forms of Discovery

There are three main avenues in AI discovery systems. They are:

- Theory-driven discovery
- Data-driven discovery
- Clustering.

These are described in the following subsections.

2.1.1 Theory- driven Discovery- AM

AM [13] is a program by Lenat that discovers concepts in elementary mathematics and set theory. An extension of AM is EURISKO [14].

AM has two inputs:

- A description of some concepts of set theory (in LISP form), e.g., set union, set intersection, the empty set.
- Information on how to perform mathematics, e.g., functions.

Given the above information, AM discovers:

Integers

- it is possible to count the elements of a set and this is an image of the counting function – the integers– interesting set in its own right.

Addition

- the union of two adjacent sets and their counting function.

Multiplication

- having discovered addition and multiplication as set-theoretic operations more effective descriptions are supplied by hand.

Prime Number

- factorization of numbers and numbers with only one factor are discovered.

Golbach's conjecture

- even numbers can be written as the sum of 2 primes, e.g., $28=17+11$.

Maximally divisible numbers

- numbers with as many factors as possible. A number k is maximally divisible if k has more factors than any integer less than k , e.g., 12 has six divisors 1, 2, 3, 4, 6, 12.

How does AM work?

AM employs many general-purpose AI techniques:

- A frame based representation of mathematical concepts.
 - *AM can create new concepts (slots) and fill in their values. For example, the AM concept learner for prime numbers may have slots like name, definition, examples etc.
- Heuristic based search.
 - *AM uses 250 heuristics that represent hints about activities that might lead to interesting discoveries.
 - * Heuristics as to how to employ functions to create new concepts, generalizations etc.

- Hypothesis and test based search. Generate-and-test is used to form hypothesis on the basis of a small number of examples and then to test the hypothesis on a large set to see if they still appear to hold.
- Agenda control of discovery process. When the heuristics suggest a task, it is placed on a central agenda, along with the reason that it was suggested and the strength with which it was suggested.

2.1.2 Data-driven discovery

There are a number of discovery systems, which are confronted with data from the real world and explore the numerical regularity among the data points. The vast majority of these systems use a combination of three searches. They occur in spaces of (1) terms of increasing complexity, (2) pairs of terms, or more generally tuples of terms, and (3) equations for pairs (tuples) of terms. The equations are a product of search (3), which typically uses least square fitting applied to a limited number of polynomial models. Search (1) transforms the initial variables to $\log(x)$, $\exp(x)$, $x \times y$ and the like. New terms are combined by search (2) in pairs and passed on to search (3) that combines them into equations and fits the best values of numerical parameters in those equations. Some equation finders have reached very high level of quality. Their main advantage over humans is breadth of search and unbiased evaluation of many equations. BACON and FAHRENHEIT play the pioneering role in these types of exploration of experimental data. So we will deal with these two systems here.

2.1.2.1 Discovering Empirical Laws Using BACON

Langley *et al.*[12] translated theories about information processing into running computer programs. This had led to a sequence of computer programs collectively called BACON. The BACON systems (versions 1 through 6) are named after Francis Bacon, because they incorporate many of his ideas on the nature of scientific reasoning. The successive versions of BACON share a common approach to discovery, as well as a common representation of data and laws. The differences among the various systems lie in the discovery heuristics that each uses in its search for empirical laws.

BACON.1 is the simplest of the systems and thus the easiest to describe and to understand. BACON.1 uses a general representation and a few heuristics to discover an impressive range of empirical laws. The system is general in the sense that the basic methods of BACON make no

reference to the semantic meaning of the data on which they operate and make no special assumptions about the structure of the data. We have striven for generality in BACON because we wish to explore the role in scientific discovery of heuristics that may be relevant over a wide range of scientific disciplines and hence may contribute to our basic understanding of discovery wherever it may occur. Having clarified our bias toward general mechanisms, let us consider an example of how one might employ such mechanisms to discover an empirical law.

2.1.2.1.1 A Sample Protocol

In 1618, Johannes Kepler discovered his third law of planetary motion: the cube of a planet's distance from the Sun is proportional to the square of its period. This law can be restated as $D^3/P^2 = C$, where D is the distance, P is the period, and C is a constant.

How might one discover such a law? Here is a sample protocol that draws on three very simple heuristics:

- If the values of a term are constant, then infer that the term always has that value.
- If the values of two numerical terms increase together, then consider their ratio.
- If the values of one term increase as those of another decrease, then consider their product.

The value of these heuristics can best be seen in their operation. The three planets considered below, A, B, C, obey a version of Kepler's law, where the constant is 1. The discoverer must begin by gathering some data, selecting different values for the nominal variable (planet), and obtaining the values of the numerical terms (D and P):

When planet is A

What is P ? Answer: 1.0

What is D ? Answer: 1.0

When planet is B

What is P ? Answer: 8.0

What is D ? Answer: 4.0

When planet is C

What is P? Answer: 27.0

What is D? Answer: 9.0

D increases with P.

So I'll consider their ratio.

I'll define Term-1 as the ratio of D and P $[D/P]$.

Here the second heuristic has been applied. The distance and the period have been observed to increase together, so the new concept Term-1 has been defined as their ratio. Next, the values of this new term are calculated.

When D is: 1.0 4.0 9.0

and P is: 1.0 8.0 27.0

Term-1 is: 1.0 0.5 0.333

D increases as Term-1 decreases.

So I'll consider their product.

I'll define Term-2 as the product of D and Term-1 $[D(D/P) = D^2/P]$.

When D is: 1.0 4.0 9.0

and P is: 1.0 8.0 27.0

Term-2 is: 1.0 2.0 3.0

Term-1 increases as Term-2 decreases.

So I'll consider their product.

I'll define Term-3 as the product of Term-1 and Term-2 $[(D/P)(D^2/P) = D^3/P^2]$.

By this point, the third heuristic has been applied twice. Two more concepts have been defined: Term-2 as D^2/P and Term-3 as D^3/P^2 . Since the latter of these is the most recently formed, we next examine its values:

When D is: 1.0 4.0 9.0

and P is: 1.0 8.0 27.0

Term-3 is: 1.0 1.0 1.0

Term-3 has the constant value 1.0.

Finally, the first heuristic applies, for the new concept Term-3 (defined as D^3/P^2) has the constant value 1.0 for all three planets. The statement that this term is constant across planets is equivalent to Kepler's third law of planetary motion, and the above protocol is a plausible trace of how one might discover this law.

2.1.2.1.2 BACON's Representation

The above protocol was actually generated by the BACON.1 program. The program represents its data in terms of *data clusters*. A data cluster is a set of attribute-value pairs linked to a common node; it represents a series of observations that have occurred together. The program knows about two types of terms or attributes: *independent* and *dependent*. It has control over independent attributes, it can vary their values and request the corresponding values of the dependent attributes. In the above example, the values of the independent term were the names of planets; the values of the dependent terms were the distances and periods of those planets. Thus in the Keplerian example, there are three primitive attributes: the planet being observed, the planet's distance D from the sun and the period P . However, much of BACON's power comes from its ability to define higher-level (theoretical) attributes in terms of more primitive ones. Thus the clusters also contain the values of three nonprimitive attributes: Term-1 (defined as D/P), Term-2 (defined as D^2/P), and Term-3 (defined as D^3/P^2). Since these terms include dependent terms in their definitions and thus cannot be manipulated by experiment or observation, they are considered dependent.

BACON is implemented in the production-system language PRISM. In turn, PRISM is implemented in LISP, a list-processing language widely used in artificial intelligence research. A production-system program has two main components: a set of condition-action rules, or *productions*, and a dynamic *working memory*. A production system operates in cycles. On every cycle, the conditions of each production are matched against the current state of the working memory. From the rules that match successfully, one is selected for application. When a production is applied, its actions affect the state of the working memory, making new productions match. This process continues until no rules are matched or until a stop command is encountered. When two or more rules match, BACON prefers the rule that matches against elements that have been added to memory most recently. This leads the system to pursue possibilities in a depth-first manner.

2.1.2.1.3 Production System of BACON – with special reference to BACON.3

As other versions of BACON, BACON.3[10] has seven major sets of productions:

1. *A set for gathering directly observable data;*
2. *A set to detect regularities in the data generated by the first and fourth sets;*
3. *A set that calculates the values of theoretical terms;*
4. *A set that checks for loops by comparing new theoretical terms to existing ones;*
5. *A set for noting related theoretical terms and ignoring their differences;*
6. *A set to collapse clusters with identical conditions;*
7. *A set for discovering irrelevant variables and ignoring their values.*

1. Gathering Data

The first set of 17 productions is responsible for gathering directly observable data. Of these productions, 7 are responsible for gathering information from the user about the task to be considered. This information consists of the names of all variables, along with suggested values for those variables under the system's control. Once this information has been gathered, the remaining 10 productions gather data through a standard factorial design.

2. Discovering Regularities

The second set of 16 productions is responsible for noting regularities in the data collected by the first set. These rules can temporarily interrupt the data gathering productions while pursuing their own goals. The system's regularity detectors can be divided into a set of *constancy detectors* and a set of *trend detectors*. Its basic constancy detector is a general version of the traditional inductive inference rule. It may be paraphrased as:

*If a dependent variable has the same value
across a number of descriptive clusters at level L
then create a cluster at level L+1 in which the variable has the value.*

After this heuristic has fired, the rule for finding conditions is applied; it is nearly as simple and

may be stated as:

*If you have just created a descriptive cluster at level L+1
based on a number of cluster at level L,
and an independent variable has the same value for all those lower level clusters,
then add that variable and its value as a condition on the new cluster.*

BACON.3's trend detectors operate only on numerical data. Some of these notice monotonic trends between variables, such as:

*If the values of the dependent variable v1 increase as the values
of the variable v2 increase in a number of descriptive clusters at level L,
then propose a monotonic increasing relation between v1 and v2 at level L,
and calculate the slope of v1 with respect to v2.*

This heuristic and a similar one for noticing monotonic decreasing relations work in conjunction with other trend detectors that further analyze the data.

3. Calculating Theoretical Values

Once a theoretical term has been defined at a given level, 3 additional productions calculate the values of this term for the clusters at that level. Once these values have been calculated, they are fairly enough for the regularity detectors and new levels of description may be created or more complex theoretical terms may be defined.

4. Noting Redundant Theoretical Terms

Before calculating the values of a new theoretical term, BACON must make sure that the term is not equivalent to an existing concept. If a redundant term's values were calculated, then mathematically valid but empirically uninteresting relationships (e.g., $x/x = 1$) could be detected. Accordingly, a fourth set of 22 productions decomposes new terms into their primitive components. If the definition of a new variable is identical to an existing definition, the term is rejected and other relations are considered.

For example, during its rediscovery of the gas law, BACON finds that the pressure, volume and the temperature are linearly related when the number of moles is unity. The slope of this line is

8.32 and the intercept is 0; accordingly two new terms are defined, the slope_{p_v,t,1} and the intercept_{p_v,t,1}. The definition of the first of these is PV/T, while the definition of the second is PV - 8.32T.

5. Ignoring Differences

Suppose BACON has defined two intercept concepts, as in the above example. The values of the first, intercept_{p_v,t,1} are 0 when the number of moles is 1, while the values of the second, intercept_{p_v,t,2} are 0 when the number of moles is 2. One would like BACON to generalize at this point, stating that the intercept of *all* lines relating the pressure-volume to the temperature is 0, regardless of the number of moles. However, because the two intercepts are different terms, the constancy detector described above cannot be applied.

BACON's solution to this problem is to note that the definitions of the two intercepts differ only by a constant coefficient, and to define an *abstraction* of the two which ignores this difference. The single production responsible for this may be stated as:

*If you have a level L slope or intercept of the variable v1
with respect to the variable v2 with a parameter p1,
and you have a second level L slope or intercept
relating these variables with a different parameter p2,
then define an abstracted slope or intercept of the v1
with respect to the v2 at level L+ 1.*

6. Collapsing Clusters

When a constancy is noted, a higher level description is created and conditions are found for it. Later, if a constancy is observed on a different variable, a separate cluster is specified. If the two clusters have identical conditions, they are combined into a single structure; only 3 productions are devoted to this process. Once this has happened to a number of cluster pairs, the values of the dependent terms can be compared and regularities may emerge.

For example, suppose BACON.3 has run experiments with a pendulum at various locations and found that Galileo's pendulum law, $P^2/L = K$, holds at each location. In this equation, P is the period of the pendulum and L is the length of the support. However, suppose that the value of K differs at each location. Now, imagine that BACON.3 drops a set of objects at each location, and finds that the acceleration of these objects also differs according to the location. Upon combining the information acquired at identical locations, a regularity is detected. Since the acceleration increases as the P^2/L increases, the product $a \cdot P^2/L$ is considered, where a is the acceleration; the value of this term is constant regardless of the location.

7. Handling Irrelevant Variables

To deal with situations involving irrelevant variables, BACON.3 draws on a set of 8 productions. The most important of these notes clusters in which the level of description for a variable's value is two more than the level at which that variable was defined; this implies that the variable most recently varied has had no effect on the dependent values. The rule may be paraphrased as:

*If a descriptive cluster at level L has just been created
to explain some clusters at level $L - 1$,
in which the values of the dependent variable $v1$ were constant,
and the level of $v1$ is $L - 2$,
and the values of the independent variable $v2$ differ in the level $L - 1$ clusters,
then stop varying the values of $v2$ since they are irrelevant.*

2.1.2.1.4 A Summary of BACON's Discovery

In summary, BACON gathers data in a systematic fashion, varying one term at a time and observing its effects. If a variable has no effects, it is marked as irrelevant and its manipulation is abandoned. If one variable does influence another, a new theoretical term is defined, incorporating both the independent and the dependent variables. If the term has not been considered before, its values are computed and examined. When these values are constant, BACON creates a new, higher level description which it treats as data on that level. The new cluster may be combined with others if it has identical conditions. When the values of the new term are not constant, it is used to define a more complex term and the process repeats. In addition, the search for useful

theoretical terms and constancies occur anew at each level of description. Taken together, these heuristics make BACON a powerful and efficient discovery system.

2.1.2.1.5 BACON at work

The mechanisms described above enable BACON to rediscover a number of laws from the history of physics. These are given in Table 2.1.

Boyle's law	$PV = k_1$
Kepler's third law	$D^3/P^2 = k_2$
Galileo's law	$D/T^2 = k_3$
Ohm's law	$IL = -rI + V$
Coulomb's law	$q_1q_2/kd^2 = k_4$
Ideal gas law	$PV/nT = k_5$

Table 2.1: Physical laws discovered by BACON.

2.1.2.2 Automated Discovery using FAHRENHEIT

FAHRENHEIT [30] is the computer system that autonomously discovers empirical theories. Each theory, in the form of a system of empirical equations, is discovered in interaction with a particular setup experiment. FAHRENHEIT uses robotic equipment to make experiments. It can empirically investigate any setup experiment with which it has been interfaced through robotic hardware.

2.1.2.2.1 Setup experiment

FAHRENHEIT captures the interaction between a discovery system and a setup experiment. There are two meanings of an experiment. In the first meaning, an experiment is a particular configuration of hardware. In the second meaning, an experiment is a single cycle of interaction between the experimenter and the empirical system, which consists in controlling a particular combination of values of some variables, and measuring the response of the empirical system in terms of some variables.

Investigating an empirical system S , scientists apply manipulators such as heaters and burettes to create the states of S which hold the desired values of the control variables. Then they use sensors such as thermometers and p^H -meters to measure responses of S in terms of variables such as temperature and p^H . Each experiment setup includes a finite, typically small, number of control and dependent variables. To set a control variable at a desired value or to measure the actual value of a dependent variable can be a complex task, requiring many simple actions of sensors and manipulators. The interaction of a discoverer with a setup experiment is two-way, when measurements are preceded by manipulations. But when the control actions are not possible or not used, the interaction is one-way and the only variables are those observed.

2.1.2.2.2 Empirical space

Consider M control variables x_1, x_2, \dots, x_M and N dependent variables y_1, y_2, \dots, y_N . Each variable $x_i, i = 1, 2, \dots, M$ is limited in scope to a set of values X_i . Each variable $y_i, i=1,2,\dots,N$ is limited in scope to a set of values Y_i . When it does not matter whether a variable is control or measured, we will use the notation z or $z_i, i = 1, \dots, M + N$, and the corresponding sets of values: Z and Z_i . The values of all variables form a Cartesian product E of $M + N$ dimensions. Each Z_i is a segment of real numbers between a minimum and a maximum value.

The values of each variable carry empirical error. In each set of values Z_i , the pairs of values are not distinguishable if they differ by less than ϵ_i . The values of ϵ_i can vary over Z_i . The values of error can be determined in the course of experimentation.

In order to make experiments, an autonomous explorer must be able to control the values of all control variables in E and be able to measure the values of all dependent variables. Experiments are the only way for obtaining information about E . Each experiment consists of enforcing a value for each independent variable $x_i, i = 1, 2, \dots, M$, and of reading the values of $y_j, j = 1, 2, \dots, N$ from the measuring instruments. The meaningful differences in values of $x_i, i = 1, \dots, M$ are not smaller than error ϵ_i .

2.1.2.2.3 Regularities

We concentrate primarily on knowledge expressed by *equations*, but it is useful to view them in the broader perspective of all regularities. A regularity holds in E if some tuples in E do not occur in any experiment. They are logically possible, but physically impossible. We can learn about what is feasible and what is not by repeated experimentation. Some outcomes occur repeatedly, while others may never happen.

Those events in E, which physically occur, may form patterns, which can be described in various languages, for instance in the form of logic statements or mathematical equations. Each concrete pattern may have a limited range of applicability, outside of which other patterns apply. Therefore the regularities take on the form:

Pattern P holds in range R

Equations are suitable for highly repeatable functional relations for numerical attributes. They typically allow deterministic predictions, excluding all except one of logically possible values. Equations are also useful in expressing quasi-functional relationships when the actual values are spread within a limited variance; for instance, the dependence between human age, weight and height. FAHRENHEIT discovers knowledge in the form of equations that hold in topologically coherent regions in a numerical space. It also seeks equations as boundary descriptions of those regions.

2.1.2.2.4 Exploration of empirical space

The task of an autonomous empirical discoverer is to generate as complete a theory of E as possible. The theory should be empirically adequate to the data, as much as possible, preferably within empirical error. The theory may include regularities between control variables and dependent variables or regularities between dependent variables. The theory may include boundary conditions for each regularity, which are typically regularities on control variables, but can also involve dependent variables.

As an example, depicted in Figure 2.1, consider a space of two independent variables x_1 and x_2 (and one dependent variable y , not shown for simplicity). Regularities R_1 , R_2 , and R_3 cover the whole range $X_1 \times X_2$ of x_1 and x_2 . Regularities are divided by boundaries B_1 , B_2 and B_3 . Figure 2.1 can be interpreted as a phase diagram, for instance, for water. Under this interpretation, R_1 , R_2 ,

and R_3 are state equations for ice, water and steam; the boundaries capture knowledge about the conditions for melting/freezing and evaporation/condensation; while β indicates the triple point of water in which steam, water and ice are in equilibrium.

All facts obtained by experiments, and all pieces of the discovered knowledge are added to the regularity-network, right after they are discovered. The network grows from the initial empty state to the complete theory of empirical space.

Figure 2.2 and 2.3 show an instance of a reg-net and its growth. In figure 2.3, each of the three state equations from figure 2.1 is represented by a node. The areas in which the equations hold are described by nodes B_1 , B_2 , and B_3 , which represent boundary equations. The boundaries are expressed in terms of independent variables x_1 and x_2 . The regularity network that represents the complete knowledge of the phase space in figure 2.1 has been depicted in figure 2.3. Figure 2.2 illustrates an intermediate state of the discovery process, after one regularity (R_1) and its boundaries have been found.

Reg-net can include nodes of higher dimensionality, which represent equations in more than two variables. Those nodes are produced by combination of several 2-D nodes. Each k -dimensional regularity node is a structure consisting of the following entries:

1. *k-dimensional pattern (an equation or a dataset)*
2. *pointers to (k - 1) - dimensional regularity nodes (boundaries)*
3. *pointers to regularity nodes in k+1 dimensions
or pointer to the list of variables yet to be considered*

After constructing the regularity network, FAHRENHEIT tries to find the regularity in two dimensions. A search for bivariate patterns becomes an active goal when enough data have been collected for two variables. In typical experiments, one of the variables is control and one dependent, while the data about a boundary typically include two control variables. When data come from observation, both variables are dependent/ measured. Typically the error is linked to dependent variables, but also possible are situations in which no error is known, or error is known for each of the variables.

The Equation Finder (EF) module of FAHRENHEIT differs from other equation finding mechanisms in several ways. It uses experimental error in a systematic, statistically sound manner.

It has been implemented with a particular concern towards a broad scope of equations it can detect, modularity of design and flexibility of control. It will be discussed in detail in section 2.2.

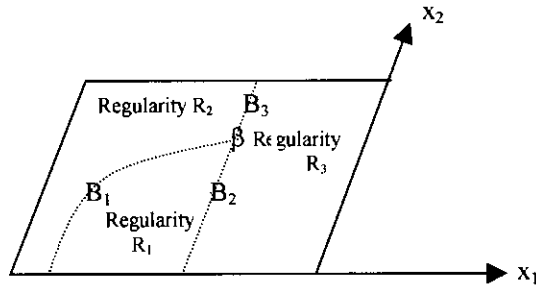


Figure 2.1: Three regularities separated by boundaries.

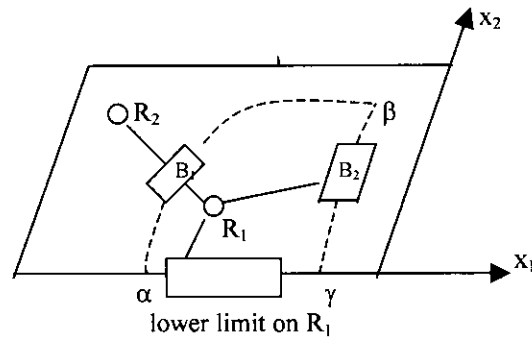


Figure 2.2: The intermediate state to discover some regularities and their boundaries.

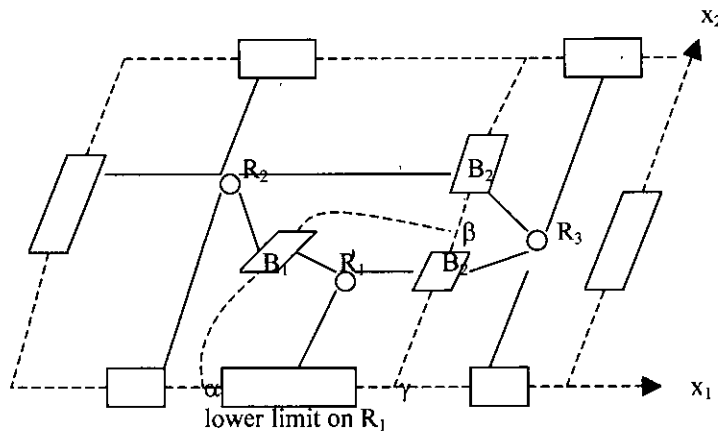


Figure 2.3: The final state of knowledge.

2.1.2.2.5 Boundary Search

After a pattern P has been detected, the full range in which P holds is still not known. For a given pattern P in E , a boundary of P is each hyper-surface such that pattern P holds on one side of the surface, while it is not satisfied on the other side. A boundary of P has dimensionality one less than the range of P . For a bivariate pattern, the range is one-dimensional and the boundaries are two points: the upper and the lower boundary.

When the search for the upper boundary for the pattern $P(x,y)$ is selected, new data must be collected and confronted with $P(x,y)$, until the boundary is found. The search splits into two phases. In the first phase the data collection follows a sequence: $x_{i+1} = x_i + x_increment$, where $increment$ is larger than the error of x , and such that a small number of experiments suffice to cover the range of x . The search follows the algorithm:

```
x := x-max ; x-max is the largest value known to satisfy P(x,y)  
LOOP  
  x := x + x_increment  
  GET y by experiment or from lower level pattern generated for x  
  IF P(x,y) ; if pattern P is satisfied by data (x y)  
    THEN add (x y) to the DATA schema associated with regularity P(x,y)  
    ELSE create a SEEDS schema of the type DATA  
      add (x y) to the SEEDS schema  
      RETURN from LOOP  
END LOOP
```

After the first piece of data has been found that contradicts pattern P , the linear search terminates. It is followed by the binary search aimed at narrowing the gap between the known points which satisfy the pattern and those which do not, until the difference is not larger than the error for x :

```

x1 := max value which satisfies P(x,y)
x2 := min value which does not satisfy P(x,y)
LOOP UNTIL  $|x2-x1| \leq \text{error\_of\_x}$  AND RETURN  $(x1+x2)/2$  as the boundary value
  x :=  $(x1+x2)/2$ 
  GET y by experiment or from lower level pattern generated for x
  IF P(x,y)
    THEN add (x y) to the DATA schema associated with regularity P(x,y)
      AND x1 := x
    ELSE add (x y) to the SEEDS schema
      AND x2 := x
END LOOP

```

The search returns x_0 such that $P(x, y)$ is satisfied for $x \approx x_0 - \epsilon/2$, while not satisfied by $x \approx x_0 + \epsilon/2$.

The search for the lower boundary follows the analogous schema.

The boundary search works in the same way for patterns of any dimension. For multi-dimensional patterns, however, instead of single experiments, the values of parameters in the patterns are determined as numerical values of coefficients in equations discovered for the smaller number of dimensions. A pattern may include not only an equation in k variables, but also its boundaries, in the form of equations in $k - 1$ variables.

2.1.2.2.6 Generalization of empirical regularities

After a k -dimensional regularity has been detected, one of the goals is to expand the regularity to $k+1$ dimensions. FAHRENHEIT uses BACON's generalization mechanism [11] and expands it from regularities to boundaries and all types of patterns discovered in data. The search in $(k+1)$ -th dimension follows the same steps as discovery of regularities in one dimension. As in BACON, role of facts, obtained at the lowest level by direct experiments, is played at $(k+1)$ -th level by parameter values for patterns discovered at k -th dimension.

Generalization to a control variable x_k starts from data collection. A sequence of values of x_k is generated according to the schema analogous for every dimension: $x_{i,k} = x_{\text{init},k} + (i - 1) \times I_k$, $i =$

1,2,... where I_k is the increment of x_k , and x_{init} is a default initial value, such as room temperature or normal atmospheric pressure. All other control variables which have not yet been varied are kept constant. For each value of x_k , the whole $k - 1$ dimensional search is repeated. If successful, it creates a $k - 1$ dimensional reg-net similar to the original network.

2.1.3 Clustering

In inductive learning, a program learns to classify objects based on the labelings provided by a teacher. In clustering, no class labelings are provided. The program must discover for itself the natural classes that exist for the objects, in addition to a method for classifying instances.

AUTOCLASS [1] is one program that accepts a number of training cases and hypothesizes a set of classes. For any given case, the program provides a set of probabilities that predict into which class(es) the case is likely to fall. In one application, AUTOCLASS found meaningful new classes of stars from their infrared spectral data. This was an instance of true discovery by computer, since the facts it discovered were previously unknown to astronomy. AUTOCLASS uses statistical Bayesian reasoning for its operation.

2.2 Discovery of Equations

All empirical discoverers use modules that find mathematical equations in two variables from a sequence of data. A measurement error is associated with each experimental data point, and that all knowledge derived from experimental data carries corresponding errors. Systems like BACON, IDS[19] and ABACUS [2] disregarded or oversimplified error analysis or error propagation. The equation finder (EF) module of FAHRENHEIT handles experimental error in a statistically sound manner. The equation finder will now be discussed in the following subsections.

2.2.1 Model fitting and evaluation

Input to equation finder consists of N numeric data points (x_i, y_i, σ_i) , $i = 1, 2, \dots, N$, where x_i are the values of the independent variable x , y_i are the values of the dependent variable y , σ_i represents the uncertainty of y_i (scientists call it *error*, for statisticians it is *deviation*, while the term *noise* is often used in AI). Output is a list of acceptable models.

Chi-square fitting, known also as *weighted least-squares*, is used to fit given numeric data points (x_i, y_i, σ_i) to a finite number of models. Model is a function template $y = f(x, a_1, \dots, a_q)$ (for example, $y = a_1 + a_2x + a_3x^2$) whose parameters' values a_1, \dots, a_q are determined by the requirement that the value of χ^2 ,

$$\chi^2 = \sum_{i=1}^N \left(\frac{y_i - f(x_i, a_1, \dots, a_q)}{\sigma_i} \right)^2 \quad (1)$$

is minimal. The value of χ^2 is the sum of squares of deviations of data points (x_i, y_i) from the model, weighted by errors σ_i , so that measurements which are more precise carry greater weight. At the minimum of χ^2 , its derivative with respect to the a_j all vanish,

$$\sum_{i=1}^N \frac{y_i - f(x_i, a_1, \dots, a_q)}{\sigma_i^2} \cdot \frac{\partial f(x_i, a_1, \dots, a_j, \dots, a_q)}{\partial a_j} = 0, \quad (2)$$

for $j = 1, \dots, q$.

In general, the set (2) of equations is non-linear and not easy to solve. For a polynomial model, however, the set (2) can be solved by algebraic or matrix operations, producing efficiently a unique solution.

Standard deviations of parameters a_1, \dots, a_q at the values that minimize χ^2 are calculated according to the formula for error propagation :

$$\sigma_{a_j}^2 = \sum_{i=1}^N \left(\frac{\partial a_j}{\partial y_i} \right)^2 \cdot \sigma_i^2, \quad j = 1, \dots, q \quad (3)$$

It is to note here that $a_j, j=1, \dots, q$ are solutions of equation (2), therefore they are functions of x_i, y_i, σ_i .

Sometimes, there is a need for the removal of some parameters. From Figure 2.4, it is obvious that there may be three models for the plotted data: $y = a_1 + a_2x + a_3x^2$, $y = b_1 + b_2x + b_3x^2 + b_4x^3$, and $y = c_1 + c_2x + c_3x^2 + c_4/x$; b_4 and c_4 are "zero-valued": $|b_4| \ll \sigma_{b_4}, |c_4| \ll \sigma_{c_4}$. Note that (i) there is no visible difference between the first two models; (ii) the data does not prefer any of the models, all have the same goodness of fit defined by (1), but $y = a_1 + a_2x + a_3x^2$ is the simplest. Higher degree polynomials will always be created, it is important to eliminate those which are unnecessary.

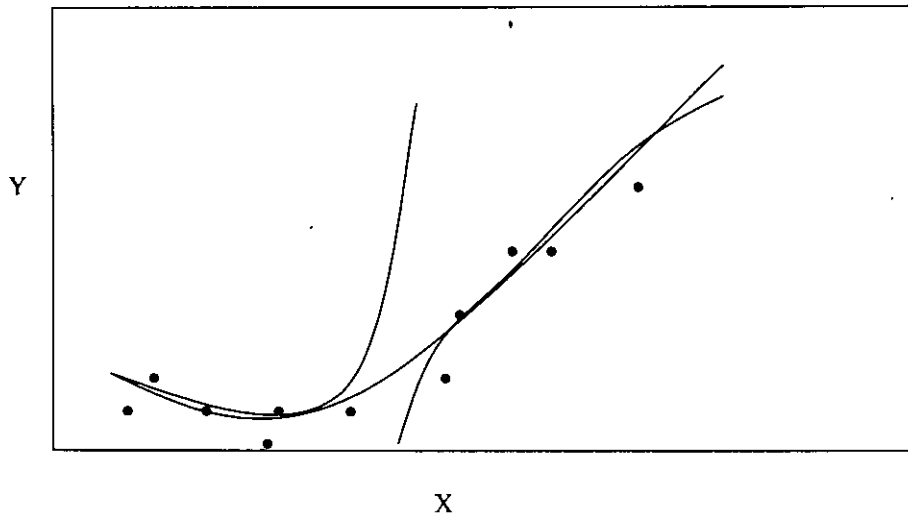


Figure 2.4: Sample data and 3 equations with similar goodness of fit

For the best fit to each model, we have to calculate the value of χ^2 according to equation (1), and assign the probability $Q = Q(\chi^2, N - q)$ ($N - q$ is the number of degrees of freedom in the data left after the fit) that this χ^2 value has not been reached by chance. A small value of Q means that the discrepancy between the data (x_i, y_i, σ_i) and the model $y = f(x, a_1, \dots, a_q)$ is unlikely to be a chance fluctuation. The *threshold of acceptance* is defined using the probability Q : all models for which Q is smaller than some minimum value Q_{\min} , are rejected. The best models are those with the largest values of Q .

2.2.2 The Search Space of the Equation Generator

The class of polynomial models is too narrow for many applications. To extend the scope of detectable functions, the equation finder uses data transformations, combined with application of polynomial model fitters to the transformed data (figure 2.5). For each pair of variables x', y' , and the error σ' of y' , Equation Generation Search (left hand side of figure 2.5) creates new variables by applying a number of transformation rules, initially to the original variables x and y ,

$$x \rightarrow x' = t_1(x) \quad (4)$$

$$(x, y, \sigma) \rightarrow (y', \sigma') = \left(y'(x, y), \left| \frac{\partial y'}{\partial y} \right| \cdot \sigma \right), \quad (5)$$

and then the transformed variables. We note that each time y is used in a transformation, the associated σ is also transformed according to (5). Each new term is simplified and then compared against all the previous terms to ensure its uniqueness.

The default set of transformations includes logarithm, exponent, inverse, square root, multiplication, and division. Adding new transformation rules is simple. Only the transformation formula must be provided; the formula for error transformation is automatically developed by the module that calculates analytical derivatives. For example, the inverse and multiplication transformations are defined by

name:	INVERSE	PRODUCT
parameters:	(p)	(p q)
formula:	(/ 1p)	(* p q)
application condition:	p≠0	-

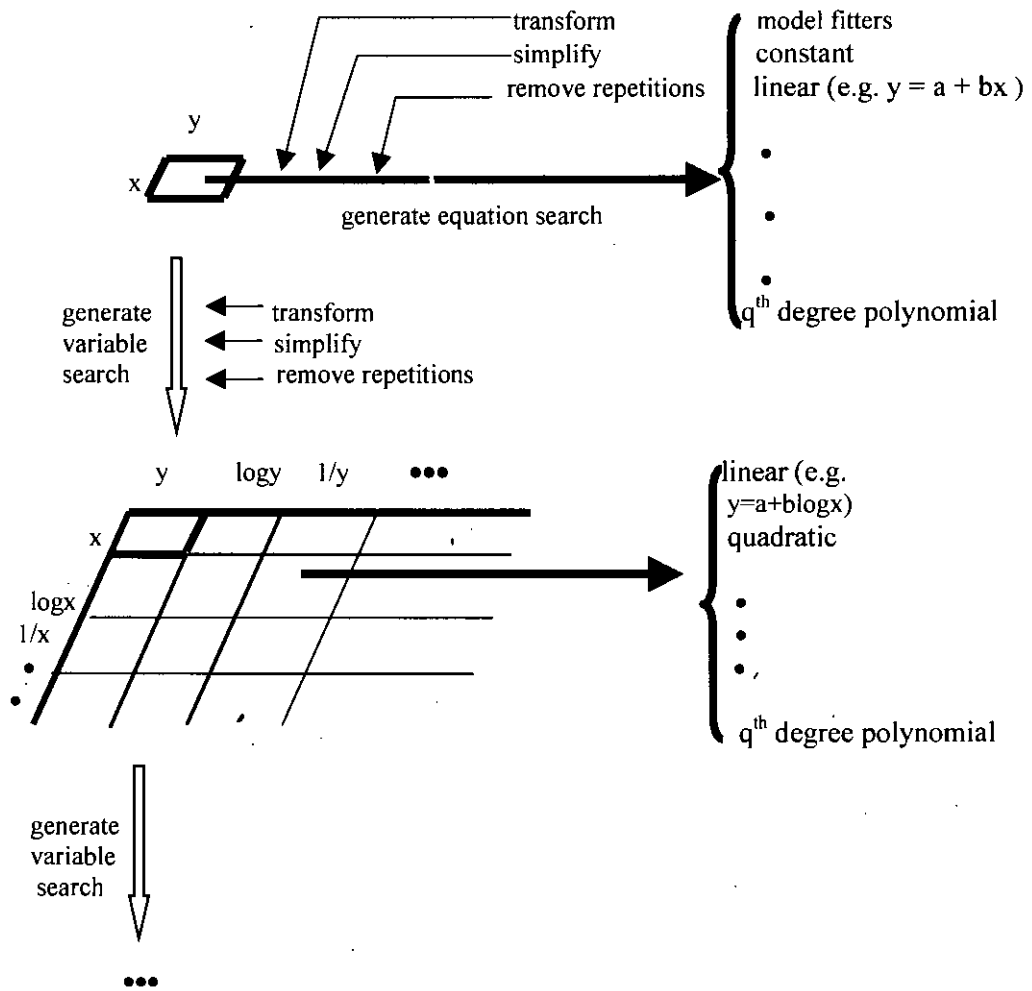


Figure 2.5: The search space: generation of variables, generation of equations, and model fitting.

2.2.2.1 Equation Generation Search

For each possible combination of variables x' and y' , and for each polynomial up to the maximum user-specified degree, the operator Generate New Equations proposes a polynomial equation $y' = f(x', a_1, \dots, a_q)$, which is then solved for y , if possible. If this seems a new equation, the Model Fitter finds the best values for the parameters a_1, \dots, a_q , and then evaluates the model.

2.2.2.2 The Search Driver

The search driver coordinates the search in the spaces of variables and equations (Figure 2.5). The system applies the Generate New Variable operator to extend the space of variables, and the Generate New Equation operator to generate, fit, and evaluate equations.

The search starts from the two original variables: x and y (upper left part of Figure 2.5). The application of the Generate New Equation operator leads to a constant model $y = a$, linear model $y = a + bx$, and other polynomials up to the predefined maximum polynomial degree (upper right part of Figure 2.5). If none of the models passes the evaluation, the Generate New Variable operator is applied repeatedly (lower left part of Figure 2.5).

If we consider only three transformations: logarithm, inverse, and multiplication, then the new variables would be:

$$\log x, \quad \frac{1}{x}, \quad xy, \quad \log y, \quad \frac{1}{y}$$

Now, using the original and new variables, the Generate New Equation search applies to each combination of x -type and y -type variables would form a large number of models (lower left part of Figure 2.5). For example, if maximum polynomial degree is 1, then the following new equations will be generated:

$$\begin{array}{lll} y = a + \frac{b}{x} & y = ae^{bx} & y = a + b \log x \\ y = \frac{1}{a + bx} & y = ax^b & y = \frac{1}{a + b \log x} & y = \frac{a}{x} + \frac{b}{x^2} \\ y = \frac{x}{a + bx} & y = ae^{\frac{b}{x}} & y = \frac{a + b \log x}{x} \end{array}$$

Still none of the models is acceptable, the Generate New Variable search applies again. This operator, for the same three transformations, would generate the following new variables:

$\log(\log x)$	$\frac{1}{\log x}$	$x \log x$	$\frac{\log x}{x}$	$xy \log x$	$y \log x$	$\frac{y}{x}$
$\log(\log y)$	$\frac{1}{\log y}$	$y \log y$	$\frac{\log y}{y}$	$xy \log y$	$x \log y$	$\frac{x}{y}$
$x^2 y$	xy^2	$\log(xy)$	$\frac{1}{xy}$	$\log x \times \log y$	$\frac{\log x}{y}$	$\frac{\log y}{x}$

It is noticeable that the number of variables grows rapidly.

Figure 2.6 demonstrates the creation of new variables and equations from another perspective. It depicts the backtrace of actions needed to generate the equation

$$y = e^{(a + bx^2 + cx^4) / x}$$

This equation can be generated at depth two in the Generate New Variables search (the original variables x and y are at depth zero).

The search terminates when an acceptable model is found or at the predefined maximum search depth.

2.2.2.3 User control over search

The user can change the maximum depth of both searches, that is, the maximum polynomial degree and the maximum transformation search depth. The user can also specify which transformation rules are to be considered. Another parameter controls the minimum search depth. It can force the equation finding module to continue the search even if an acceptable model was found. This option is particularly useful when the user is not satisfied with any of the models already found and wants to deepen the search.

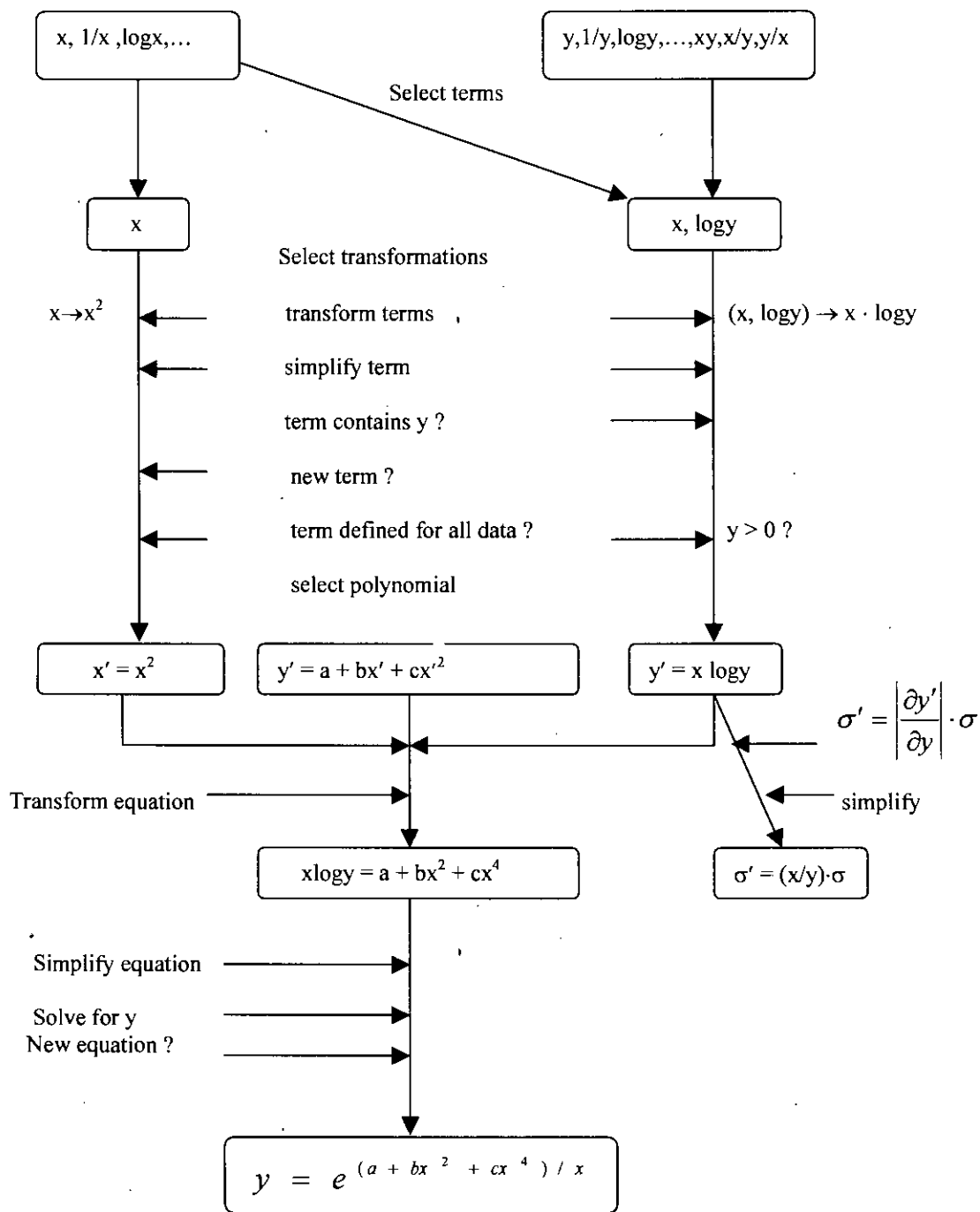


Figure 2.6: Generation of new variables and equations - example for the equation, $y = e^{(a+bx^2+cx^4)/x}$.

2.3 ABACUS: Integrating Quantitative and Qualitative Discovery

Most research on inductive learning has been concerned with qualitative learning that induces conceptual, logic-style descriptions from some given facts. In contrast, quantitative learning deals with discovering numerical laws that characterizes empirical laws. ABACUS[2] attempts to integrate both types of learning by combining newly developed heuristics for formulating equations with the previously developed concept learners.

2.3.1 The ABACUS Approach to Quantitative Discovery

There are many strategies to derive an equation or set of equations summarizing the behaviour of some physical process. In choosing a particular strategy, one must weigh the gains from the use of that strategy against the losses. The approach taken in ABACUS has been to satisfy as many criteria from our list for quantitative discovery as possible, and to reduce the user supplied information to a minimum.

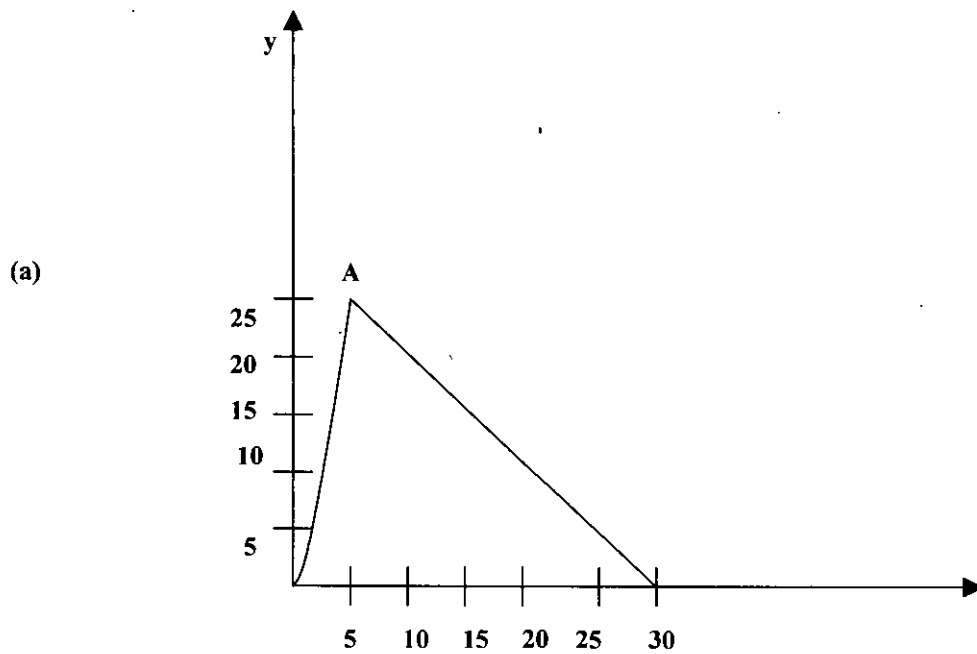
The ABACUS method of quantitative discovery consists of two steps. First, the *equation discovery module* analyzes the original empirical data and attempts to derive equations summarizing the observed behaviour. If more than one equation is required to describe the observations, the data are divided into disjoint subsets, and equations are determined for each subset. The second step passes the resulting subsets to the *precondition generation module*. This module derives a logic-style description for each subset. Such a description is used as a precondition for each equation. The result is a series of if-then rules in which the 'if part' states the precondition for applying the rule in the 'then part'.

The equation learning module searches for the best equation to describe the given data. If a single equation holds for all events, the learning task is complete. Sometimes several classes of events can be described by one expression that evaluates to different values. When this occurs, a number of classes are formed, one for each value of the expression. The following example is used to illustrate the general algorithm used in ABACUS.

2.3.2 Discovering Bivariate Equations in ABACUS

ABACUS depicts quantitative discovery as a search through the space of possible equations. This search process mathematically combines variables to form new terms. Before describing the search algorithm, we discuss how the nodes are formed, the constraints to the search and how goal is fixed.

Suppose the system is given the data depicted in Figure 2.7(a). Observed values for x and y are read in and the equation discovery module is invoked. As there are only two variables, the space of possible equations is small. The best equation found, which describes 70% of data is $x^2 = y$ (Equation formation technique is illustrated in Section 2.3.2). Events covered by the equation are put in a class associated with this equation.



- (b)
- | | | |
|----------------|-------------|------------------------------|
| Rule A: | IF | $ x = 0.10, \dots, 5.10 $ |
| | THEN | $x^2 = y$ |
| Rule B: | IF | $ x = 5.10, \dots, 30.00 $ |
| | THEN | $x + y = 30.00$ |

Figure 2.7: ABACUS analysis of graph example.

The equation discovery module is invoked again to analyze the remaining events. This time, $x + y = 30$ is found to hold for all events and a class set is created for these events. Because all observations are accounted for, the equation discovery step is completed and the precondition module is called. This module searches for properties of the data which distinguish between the two classes. The results are represented in Figure 2.7 (b). They state that when x is below 5, the equation is $y = x^2$, and when x is between 5 and 30, the equation $x + y = 30$ holds.

2.3.2.1 Variable Dependency and Proportionality Graph

At the heart of quantitative discovery is the concept that one variable's values may be dependent in some way upon the values of another variable. BACON looked for monotonic relationships in the data to create new hypotheses; i.e., the relationship between two constants are held taking all other variables to be constant. There are two problems with such a strict definition. First, for a given set of data, it is not always possible to observe changing values of x and y while holding all other variables constant. Second, we must allow for inaccuracies and errors in experimental data. As a result, we are interested in the degree with which x is proportional to y rather than detecting if x exhibits a monotonic relationship to y for all of the data. With this in mind, we say that x is *qualitatively proportional* to y if, for a given percentage of the events (user specifiable), the values of x rise when the values of y rise while certain specified variables are held constant. Similarly, x and y are *inversely qualitatively proportional* if x decreases as y rises for a majority of the events under the same conditions. There are then four assertions possible as the result of a qualitative proportionality measurement:

- Prop⁺ (x, y) - x and y are qualitatively proportional to a user-specifiable degree
- Prop⁻ (x, y) - x and y are inversely qualitatively proportional to a user-specifiable degree
- Prop[?] (x, y) - insufficient data to determine if x and y are related
- Norel (x, y) - x and y are not related

To make a qualitative proportionality assertion about variables x and y , ABACUS looks for general trends in the data. Since it is not always possible to hold all other variables constant, an *exclusion set* is defined to be the set of attributes which do not need to be held constant and is constructed by the program and the user. The user must recognize which variables simply cannot

or should not be held constant. Similarly, when measuring the proportionality between variables x and y , the program recognizes that, if x is a program generated variable composed of user defined variables v and w , then v and w should be removed from the set of variables which must be held constant. The proportionality criterion has a margin of tolerance, allowing a moderate degree of noise and a limited amount of *conflicting proportionalities*. Conflicting proportionalities occur when some of the data indicates $\text{Prop}^+(x, y)$ and some indicates $\text{Prop}^-(x, y)$.

From these proportionality assertions we may construct an undirected graph, called a *proportionality graph*. The nodes of the graph will represent variables, and the edges will indicate the presence of a qualitative proportionality relation between their incident vertices (Figure 2.8). We will construct edges for $\text{Prop}^+(x, y)$ and $\text{Prop}^-(x, y)$ relationships, and $\text{Prop}^?(x, y)$ will be treated as *Norel*. In Figure 2.8, a is proportional (+ or -) to b , but not to c .

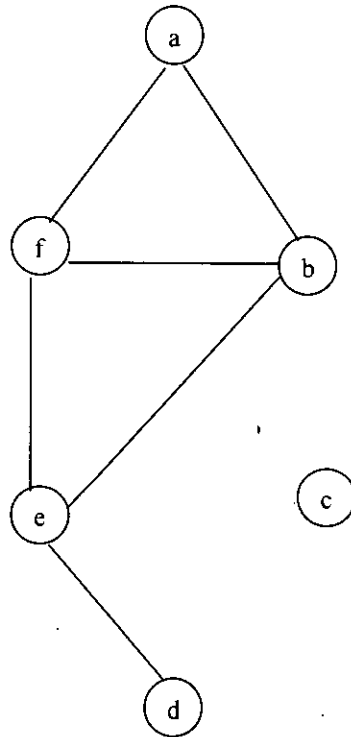


Figure 2.8: Proportionality graph

In the search algorithm, we will be concerned with cycles in these graphs. Cycles will refer to maximal cycles in a graph, i.e., cycles that are not subset of some other cycle. In figure 2.8, there is only one maximal cycle $\{ a b e f \}$.

2.3.2.2 Equation Formation - A Search For Constancy

The existence for qualitative proportionalities between variables suggests the possibility of causal or other relationships between them. For example, if we know that the value of x always goes down when the value of y goes up, then the relation $xy = \text{constant}$ might be binding these variables. This may be generalized to a rule:

If $\text{Prop}^- (x, y)$ then create a variable equal to xy

Such a variable is more likely to take on a constant value than x or y independently. Expanding on this concept, the following heuristics are formulated:

If $\text{Prop}^+ (x, y)$ then

Generate a variable equal to a quotient relation between x and y

Generate a variable equal to difference relations between x and y

If $\text{Prop}^- (x, y)$ then

Generate a variable equal to a product relation between x and y

Generate a variable equal to sum relations between x and y

With these heuristics in mind, search in quantitative discovery involves the continual combination of variables which are qualitatively proportional to form new variables in the hope of finding a variable which takes on a constant value.

2.3.2.3 Domain - independent Constraints

Several domain-independent constraints are used to limit the large search space associated with quantitative learning. These are divided into three categories:

- Units compatibility rule
- Redundancy detection
- Tautology detection

For a detailed discussion of these constraints, see [2].

2.3.2.4 Recognizing the Goal

Because a valid equation may describe only a subset of events, recognizing when a good equation has been found and when to terminate search is not as easy as it would be otherwise. There are three types of goal nodes recognized by the system. The first type corresponds to a term that describes all events, i.e., one that evaluates to the same value for every event. Such a goal is easily recognized and search terminates when one is discovered.

The second type of goal node is based on the notion of a nominal subgroup of events and also causes immediate cessation of the search process. A *nominal subgroup* is defined to be a set of events that are equal on all nominal attributes. If a term is found which evaluates to a single value for a nominal subgroup, search terminates on the assumption that an equation of significance has been found.

The third type of goal node does not halt the search algorithm. As each new variable is created, its *degree of constancy* is measured, and the variable having the largest degree of constancy is stored. The degree of constancy is defined to be the percentage of the data for which the function evaluates to a single value within a percentage range of uncertainty modifiable by the user. If search exceeds the allowed limit, the term having the highest degree of constancy is returned. If its constancy is greater than a user modifiable threshold, the resulting equation is reported. Otherwise, the program states that no formula could be found.

2.3.2.5 Search

ABACUS discovers equations by searching through the space of possible terms which relate the user supplied variables. ABACUS uses a combination of two search algorithms. The first algorithm, *proportionality graph search*, uses the graphical nature of the proportionality assertions to guide the search path and discriminate against irrelevant variables. The second algorithm, *suspension search*, enables the program to reduce the number of terms being examined by removing those that do not look promising until all other possibilities have been exhausted. We will examine the search by two examples: one is the ideal gas law, as discovered by BACON, and another is the law of conservation of momentum:

$$\frac{PV}{NT} = 8.32 \quad (i)$$

$$m_1 v_1 + m_2 v_2 = m_1' v_1' + m_2' v_2' \quad (ii)$$

2.3.2.5.1 Proportionality graph search

The proportionality graph search technique directs its search to the interrelations of variables forming a cycle and avoids variables that are not contained in a cycle. The algorithm consists of the repeated application of the following steps:

1. Form a proportionality graph for the current set of variables, both those provided by the user and those generated by the program. Exclude all edges which occurred in previously generated graphs.
2. Extract the cycles (maximal cycles) and represent each cycle by the set of nodes it contains.
3. Search each cycle in a depth-first manner for a depth given by the cardinality of the set.

The process repeats until a suitable relation is found up to a maximum of K times. The default search depth, K , is 4, since powers greater than 4 are seldom seen in the natural sciences.

For each graph, the cycle sets are sorted in decreasing order under the assumption that the largest cycles will prove to be the most promising.

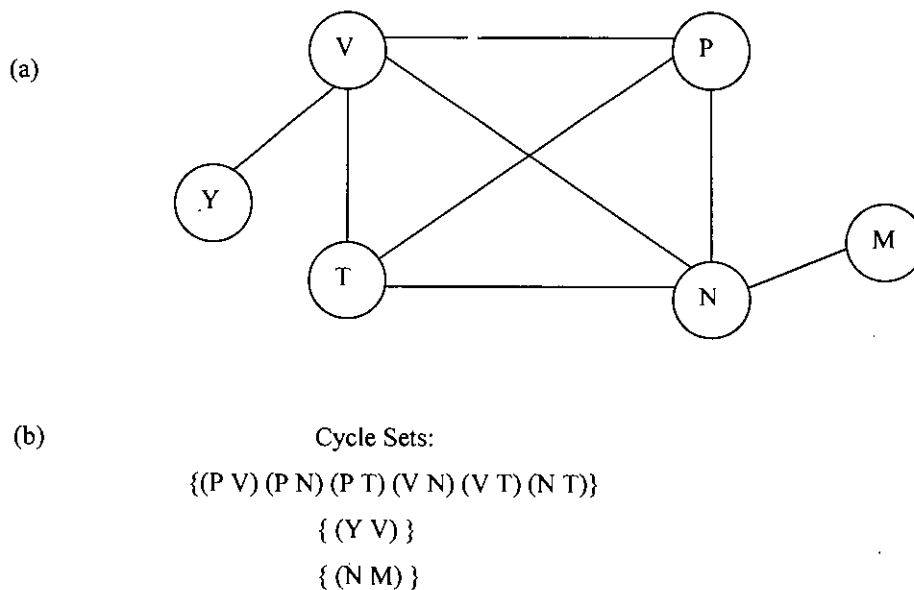


Figure 2.9: Proportionality graph search for the ideal gas law ($PV/NT = 8.32$).

A cycle (e.g. $\{V, N, P, T\}$) is searched in a depth-first manner by first removing two nodes that are proportional and combining them according to the equation formation heuristics to form new terms (e.g., V/N). The remaining nodes (e.g., $\{P, T\}$) are then tested one at a time against these terms to form new terms. For the set $\{P, T\}$ and the current node V/N , P would be tested against V/N to possibly create new terms such as PV/N . If backtracking occurred, then T would be tested against V/N . This process repeats until all combinations have been exhausted. Because nodes are removed from the cycle set as search progresses, powers of variables are not possible after the first round of search.

As an example of this search technique, a sample proportionality graph is shown in Figure 2.9(a) for the ideal gas law. A total of six attributes were initially provided by the user. The irrelevant variable mass, M , is independent of pressure, volume and temperature, but is proportional to the number of moles of gas present. A similar situation exists for variable Y .

The three cycles of the graph are given in Figure 2.9(b), where solitary edges are simply treated as a cycle having only one edge. Figure 2.10 shows the search tree resulting from the above strategy applied to the example.

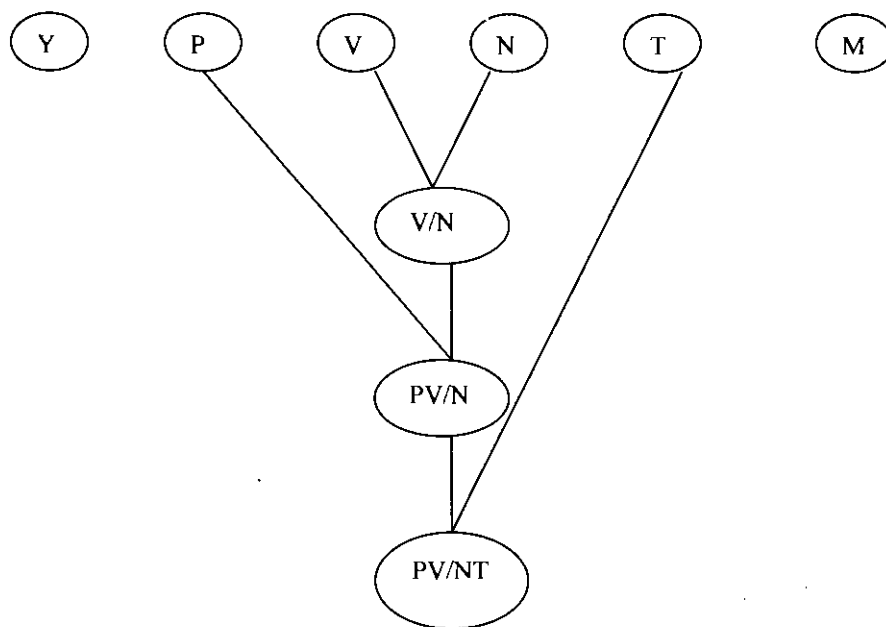


Figure 2.10: Proportionality graph search path for ideal gas law example.

2.3.2.5.2 Suspension search

To avoid the problem caused by repeated application of proportionality graph search, ABACUS uses only one iteration of the algorithm. If no law is found, then the program employs a technique called suspension search. This algorithm is able to remove nodes from consideration, yet allows their return should they be needed. Suspension search begins by normal breadth- first search. But whenever each level is created, all nodes on that level are divided into *active nodes* and *suspended nodes*. Suspended nodes are those whose constancy is less than a low threshold.

Search then proceeds on to the next level, where only the active nodes of previous levels are visible to the search algorithm. Since suspended nodes are ignored, fewer nodes are involved in the search at any one time. Therefore, search may be allowed to explore deeper than it could otherwise. A second search depth limit is defined, called the *filler depth*, which cites a limit shallower than absolute search depth. Search may proceed beyond the filler depth, but only active nodes are allowed. Suspended nodes beyond the limit are permanently discarded.

The suspension search may be summarized as:

FUNCTION Suspension(active_ancestor_nodes,active_nodes,suspended_nodes, environment)

- *If the search depth limit has been reached
then return true if the best constancy found is greater than threshold else return false*
- *If new active or suspended nodes can be created from the current list of active nodes
then return true if one of these has a constancy of 100%
or return true if a call to Suspension using the new nodes returns true*
- *If the filler depth has been reached
then save the environment and return false*
- *If new active or suspended nodes are created from the current list of suspended nodes
then return true if one of these has a consistency of 100%
otherwise save the environment
and return true if a call to Suspension using the new nodes returns true*
- *Save the environment and return false*

A partial suspension search is given in Figure 2.11 for the example involving the discovery of the law of conservation of momentum.

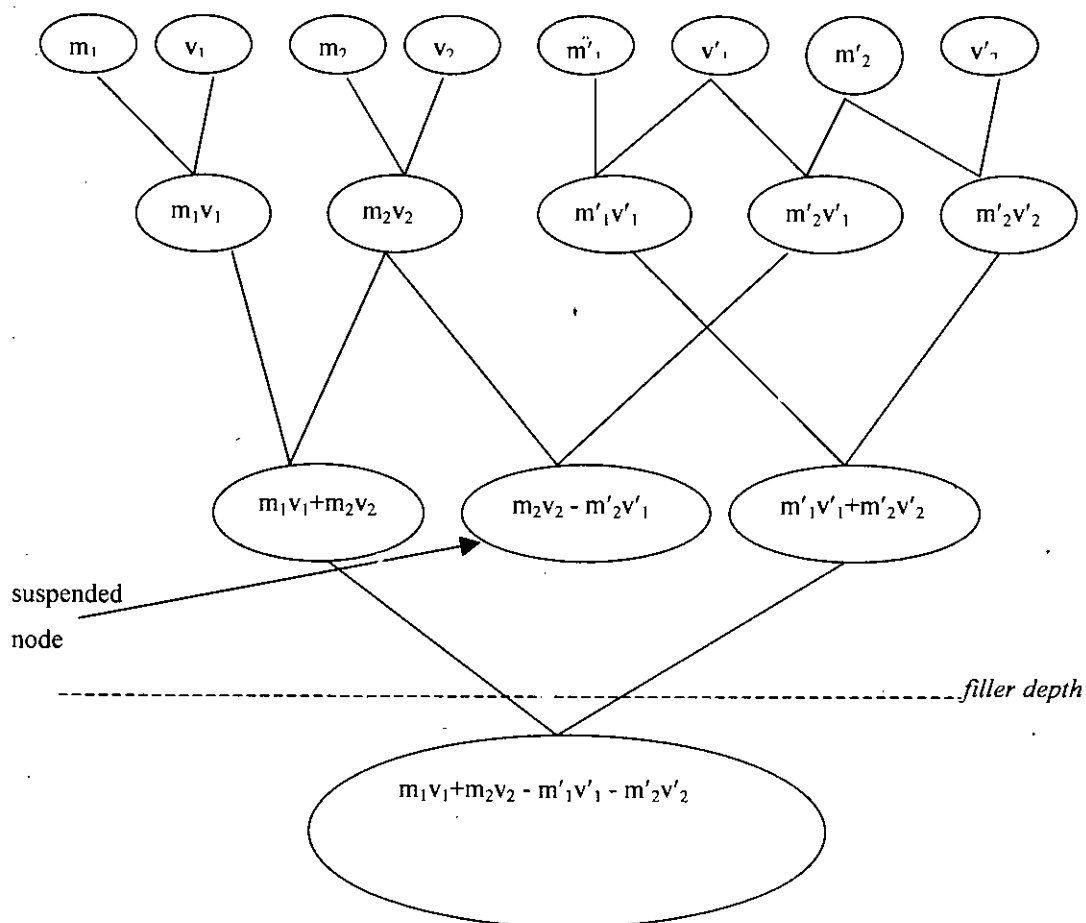


Figure 2.11: Partial suspension search tree for conservation of momentum

Combining the proportionality graph search algorithm with the suspension search algorithm favors quick discovery of laws which are composed solely of multiplication and division while still being adept at discovering more complicated equations in a reasonable amount of time. As cycles in the first pass can never be larger than the number of given attributes, the depth-first search of the first phase is not deep for most problems, thus creating variables which would normally be created for more complicated examples anyway.

Chapter 3

Development of the system

3.1 Equations and Situations

To understand the decomposition of quantitative regularities into simple expressions and their recombination into models, one must first understand the relationship between equations and physical situations. This will be described now.

3.1.1 Correspondence between Situations and Equations

Let us first examine some examples in which we compare physical situations and the corresponding equations. Consider a process in which two samples of water at different initial temperatures are combined and eventually reach thermal equilibrium (Figure 1.2a). The experimental space is spanned by four independent variables under the experimenter's control: two initial temperatures, t_1 and t_2 , and two masses m_1 and m_2 . The outcome is the final temperature t_f . Black's equation for this process can be discovered by a BACON-like system, usually in the form:

$$t_f = \frac{m_1 t_1 + m_2 t_2}{m_1 + m_2} \quad (1a)$$

Now we replace the water in the first sample by a small piece of ice at the melting temperature t_1 . The ice melts entirely in the water from the second sample (Figure 1.2b). This process is described by the equation:

$$t_f = \frac{m_1 t_1 + m_2 t_2 - m_1 c_f}{m_1 + m_2} \quad (1b)$$

where c_1 is the latent heat of the melting ice. Still another experiment, with a larger piece of ice that melts partially (Figure 1.2 c), leads to the pair of equations

$$m_{1f} = m_1 + (m_2 t_1 - m_2 t_2) \quad \text{and} \quad t_2 = t_1 \quad (1c)$$

where m_{1f} is the final mass of ice. Many similar experiments are possible, in which ice is dropped into a glass of scotch, a piece of potassium is dropped into water, water is poured over dry ice, and so forth. Each time the outcome is described by a different equation.

Other phenomena can also be varied in uncountable ways, but the same principle of composability applies. Consider a body sliding down an inclined plane, as shown in figure 1.3a. If the initial velocity is zero and the difference in height is h , then the equation that describes a class of such experiments is

$$v^2 = c_1 h \quad (2a)$$

where v is the final velocity and c_1 is a constant. Now suppose that another body is on the downward path but this time it rolls without slippage, as in figure 1.3b. The equation here will differ, assuming the form

$$v^2 = c_2 h \quad (2b)$$

in which c_2 is another constant, $c_2 = 5c_1/7$. Here it is less clear what new expressions are added to equation 2a, but when we rewrite 2a and 2b as

$$mgh = mv^2 / 2 \quad (2a')$$

and

$$mgh = mv^2 / 2 + mv^2 / 5 \quad (2b')$$

we can attribute the expression mgh to the change of altitude, $mv^2/2$ to the process of sliding, and $mv^2/5$ to the process of rolling.

The correspondence of physical components and equation parts applies not only to processes but also to state descriptions, as in the following example of electric circuits. Consider the simple circuit shown in figure 1.4a, which is described by the equation

$$E = IR + Ir \tag{3a}$$

where E is the electromotive power of the battery, r is the internal resistance of the battery, and R characterizes the resistor. By adding more elements to the circuit, as depicted in figure 1.4b, the equation becomes

$$E + E_1 = IR + Ir + IR_1 + Ir_1 \tag{3b}$$

where additional expressions E_1 and Ir_1 correspond to an additional battery, and IR_1 corresponds to the added resistor.

We have seen in these simple examples how science deals with the complexity of physical situations by combining analysis with synthesis. Scientists decompose laws into simple expressions that correspond to and are interpreted by the generic physical components of the situation. These expressions are elementary units of recombination, that is, building blocks from which complexes of equations that describe complex physical situations are recombined into models. This process allows the scientist to *transfer knowledge* extracted by analyzing simple situations to synthesize descriptions of complex situations. In most real-world situations, one must apply a combination of basic laws to generate an adequate description. However, the number of elementary components is small as compared with the total set of physical situations, and a small number of elementary expressions lets one build a limitless number of models.

3.2 The Developed System

Our system decomposes equations like the law of equation (1a) into simpler expressions. The system's input consists of an equation that describes a particular physical process P , along with a description of P in which the measured variables are attached to the appropriate elements in P . The system transforms equations until they can be interpreted by fitting a given equation and the corresponding process description. Under a satisfactory fit, all expressions that occur in the final form of the equation are assigned to the corresponding elements of the process description.

3.2.1 Representing Processes and Equations

The thermal process depicted in figure 1.2a can be represented in a process diagram (Figure 3.1) as a combination of the processes of heating and cooling, one process for each sample, coupled by energy transfer between both processes. The process diagram in figure 3.1 represents a particular understanding of the actual process. Each elementary process is associated with a particular change in a particular object.

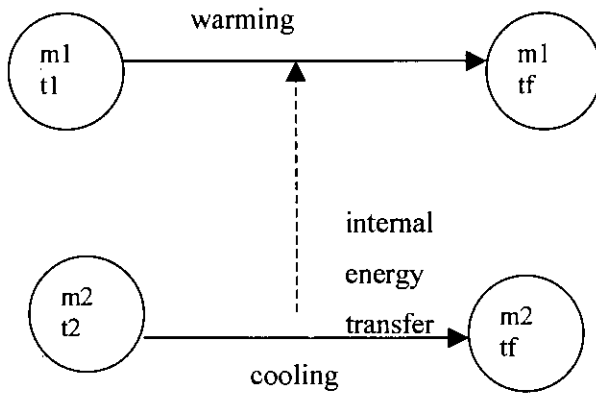


Figure 3.1: Process diagram for figure 1.2a

3.2.2 Input and Output

Figure 3.2 reviews the system's input and basic algorithm. In preparation for the main search procedure, the system represents the process diagram by a process decomposition tree like those in figures 3.3 and 3.4, and it represents the equation by a parse tree like the one in figure 3.5. A process decomposition tree offers a simplified representation of the process diagram sufficient to guide equation transformation.

One process diagram may be represented by several process decomposition trees, each providing a different perspective on the same process. For instance, figures 3.3 and 3.4 depict two different trees for the process diagram of 3.2. In figure 3.3, the process is decomposed into the initial and final states, which in turn decompose into elementary states. In figure 3.4, the same process is decomposed into two elementary subprocesses, which in turn decompose into their initial and final states. When several process decomposition trees are possible, we can operate with any of them, resulting in different possible equation decompositions.

Process diagram - obtained by observation and measurement



decomposition

Process decomposition tree



matching



Equation parse tree

*transformed until it matches
the process decomposition tree*



parsing

Equation - obtained from a BACON-like system

Figure 3.2 : Overview of the system's input (bold) and processing (italics).

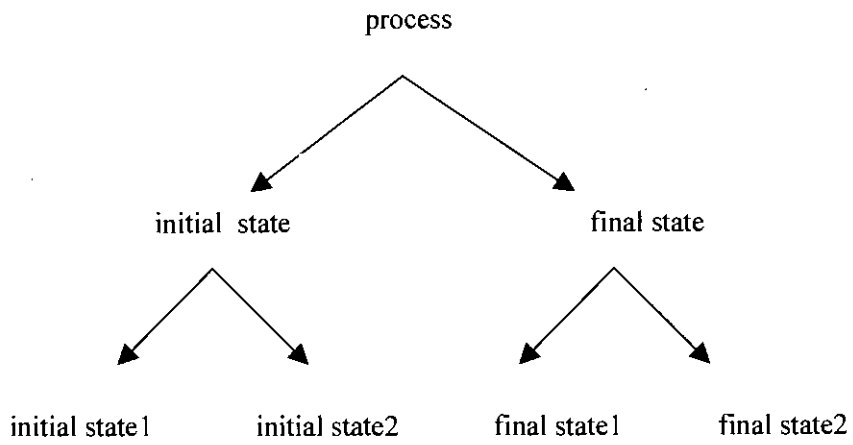


Figure 3.3: Process decomposition tree1 for the process diagram in figure 3.1

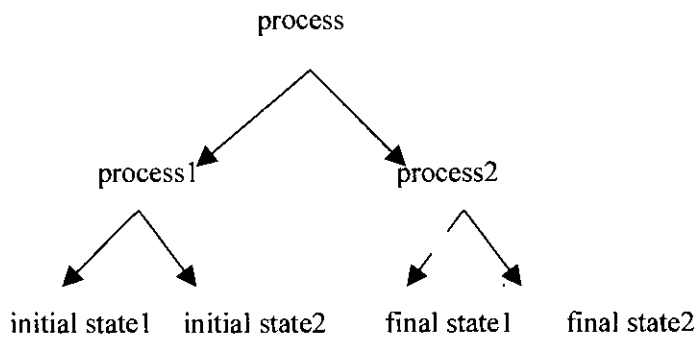


Figure 3.4: Process decomposition tree2 for the process in figure 3.1

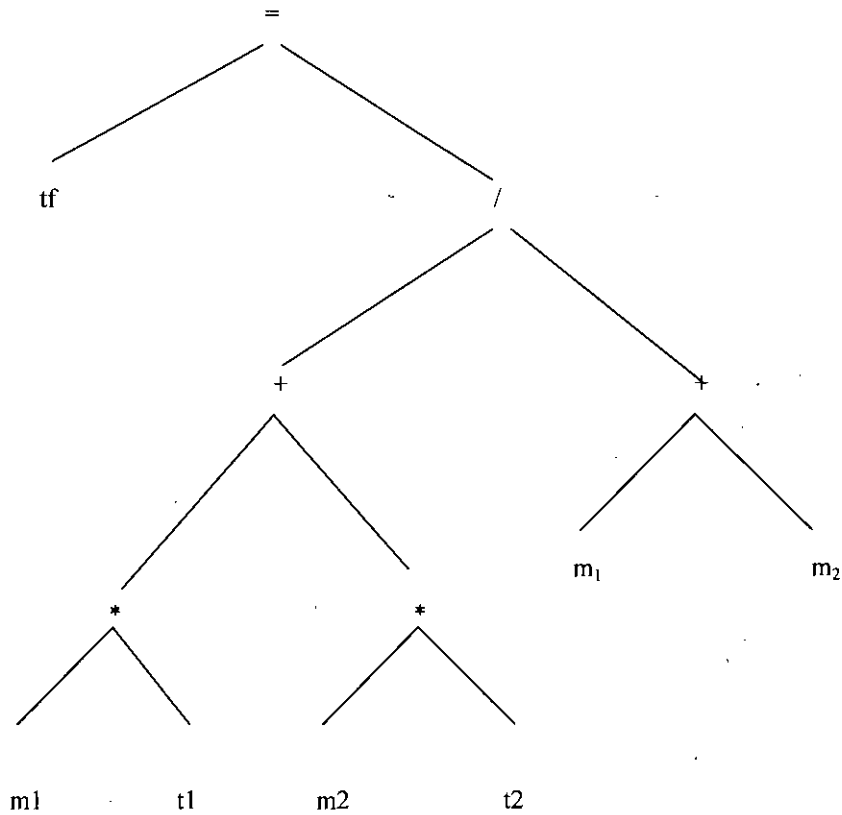


Figure 3.5: Equation parse tree for the equation $tf = (m_1t_1 + m_2t_2) / (m_1 + m_2)$.

The fitting procedure produces a mapping between the subtrees in the process decomposition tree and subtrees (expressions) in the equation parse tree. Before the search for a fit, the system is told what quantities represent measurements in each state. For instance, it is told that t_1 and m_1 describe the initial state 1, but it does not know that their product m_1t_1 is a meaningful state description. Many expressions that are built from m_1 and t_1 are potential candidates, such as t_1 , m_1 , t_1/m_1 , and $t_1m_1^2$. As a result of the fitting procedure, the system is able to identify the most appropriate expression, namely m_1t_1 , without explicit search through the space of candidate expressions. As a result of the matching procedure, the system also infers that, for the internal nodes "initial state" and "final state" in figure 3.3, the concatenation of complex thermal states is reflected by addition of the quantities that describe elementary states, giving the expression $m_1t_1 + m_2t_2$. Using the tree of figure 3.4 leads the system to determine expressions corresponding to the processes of heating and cooling: $m_1t_f - m_1t_1$ and $m_2t_2 - m_2t_f$.

3.2.3 Equation Transformation Search

To transform the input equation to an interpretable form, the equation transformation search method plays the principal role. It changes the equation to a form that matches the process decomposition tree. We use a variety of transformation grammar rules (operators) to modify the parse tree. Most of these are typical operations on algebraic expressions, which are represented in if-then form. Figure 3.6 illustrates two such rules that can be recognized as standard algebraic transformations,

$$(a + b) / c \rightarrow a/c + b/c \text{ and } a = b/c \rightarrow ac = b.$$

If the current equation or one of its parts matches the IF part of a transformation rule, it can be transformed into the structure shown in the THEN part.

Equation transformation search proceeds by applying transformation rules, which act as search operators, until it establishes a complete match with the process decomposition tree or when no improvement can be made to a partial match. It is implemented as depth-first search with backtracking, guided by the degree of match. Two other search methods work as subroutines. The first is the Match method, which tries to determine the degree of match between a given form of the equation and the process decomposition tree.

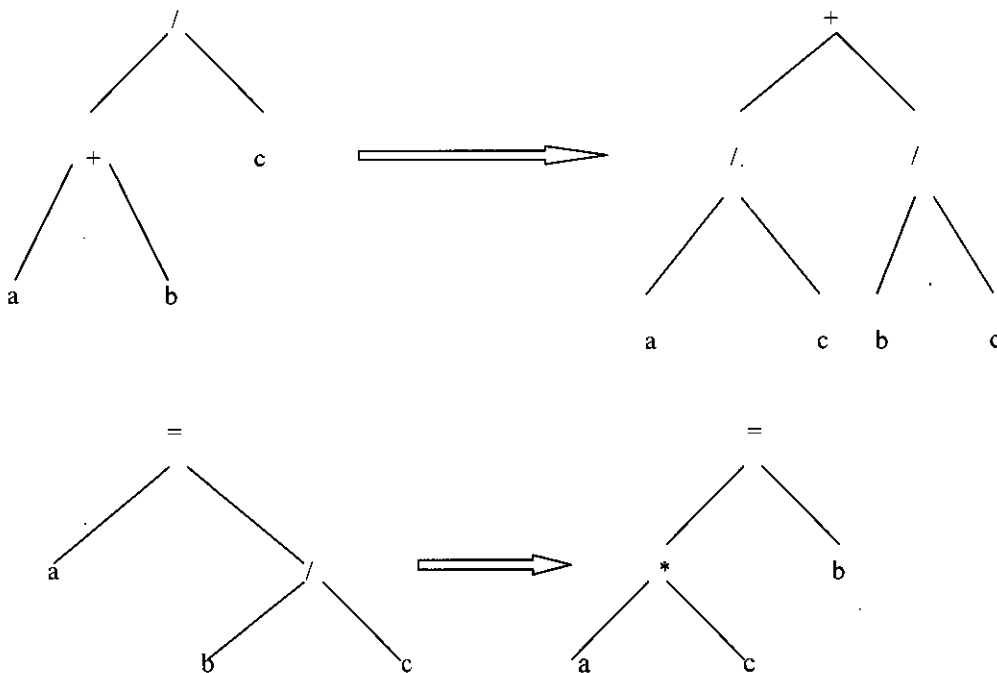


Figure 3.6: Examples of transformation grammar rules.

The second submethod, Operator Selection, uses the guidance provided by a partial match to select operators that should be applied by Transform. Operators are heuristically selected to reduce the mismatch between the equation and the process decomposition tree.

3.2.4 Tree Matching Search

The matching procedure uses the equation parse tree as its search tree, implementing a depth-first traversal of that tree with backpropagation of results. Matching starts from the leaves of the equation parse tree, where either single variables or constants are stored. A variable matches all the leaves in the process decomposition tree that are described by that variable. For instance, m_1 matches initial state 1 and final state 1, because m_1 describes each of them, but m_1 matches neither initial state 2 nor final state 2, because m_1 does not belong to the description of these states. As a result, matching at the leaf m_1 returns the list of two elements, including both initial state 1 and final state 1.

After all children of a particular node are considered, the results are backpropagated to the parent node. Internal nodes of the equation parse tree represent arithmetic operations. Depending on the arithmetic operation, one of two different list operations is applied to the lists returned from the children of a given internal node to produce the result at that node. For addition, subtraction and equality, the union is applied to the results obtained for children, whereas for multiplication and division, it is their intersection. Backpropagation stops and returns failure of the match at any node at which an empty set has been reached. When the backpropagation reaches the root of the equation, a list of physical states has been attached to each node of the equation tree that are possible candidates for the interpretation of that node.

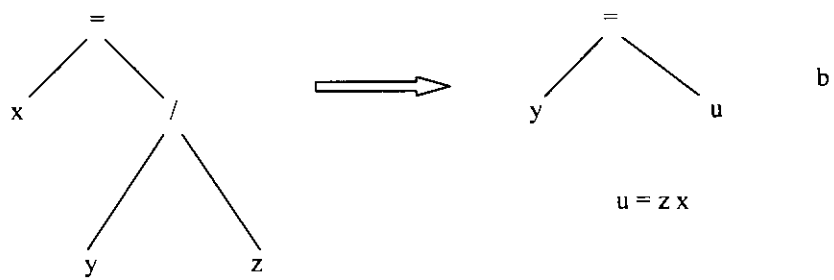
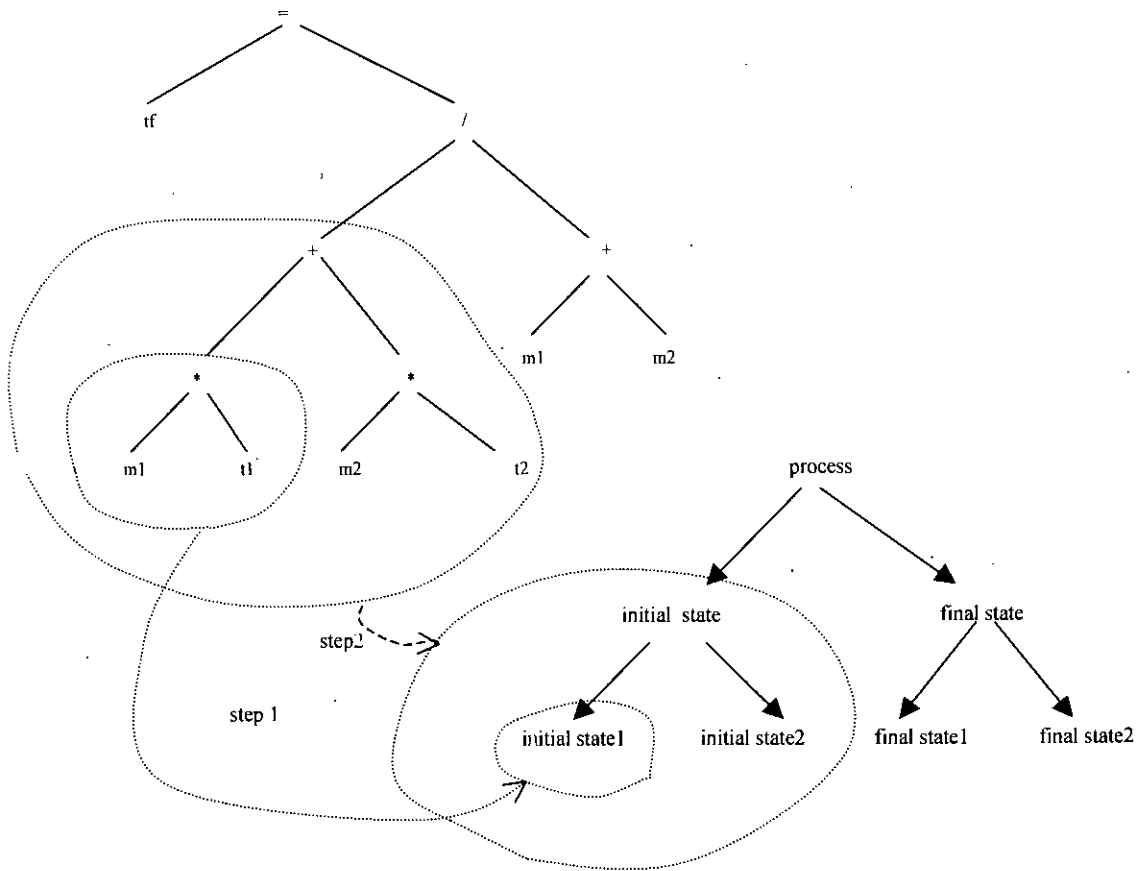


Figure 3.7. Results of matching: (a) partial match; (b) mismatch reduction schema for (a).

The next cycle of backpropagation extends the partial matches as far as possible. If the root is reached, a complete match has been detected; otherwise the backpropagation stops at a particular node when no match holds for the subtree starting at this node. If the search cannot find a total match, it uses a consistent partial match to guide operator selection for the next step of equation transformation.

3.2.5 Operator Selection

A partial match between an equation and a process decomposition tree helps us select an appropriate operator for continued equation transformation. As a result of matching the equation in figure 3.5 and the tree in figure 3.3, the system detects a partial match between the initial state and the expression $m_1t_1 + m_2t_2$, as illustrated in figure 3.7(i). Other elements of the equation in figure 3.5 do not match the tree in figure 3.3. This partial match lets the system schematically characterize the desired transformation of the equation presented in figure 3.7(ii). This transformation schema, called a *mismatch reduction schema*, is input to the search method for selecting the relevant operator transformations. This method tries to identify the rules in the transformational grammar, such as those depicted in figure 3.6, that satisfy the mismatch reduction schema.

3.3 The system with two case studies

The system developed is coded in Turbo Prolog with a large number of predicates. But if we look into the depth of it, we will find that it consists of a few subsections or modules. The modules are for process tree input, parse tree input, to find the result at the internal parse nodes, to input matches at the leaves i.e., the correspondence of the parse leaves to the nodes of the process decomposition tree, to find the matches at the internal parse nodes, accumulation of the result and process nodes at these internal parse nodes in a declarable form and then if not declarable, use of some operators in the state-space search technique to reach a goal. The operators, as has already been mentioned are equation transformation rules from algebra. In our system, we have incorporated nine most useful transformation rules. These are:

1. $a = b/c \rightarrow b = ac$

2. $b = ac \rightarrow a = b/c$
3. $(a_1 + a_2 + a_3 + \dots + a_N) / c = a_1/c + a_2/c + \dots + a_N/c$
4. $(a_1 + a_2 + a_3 + \dots + a_N) * c = a_1*c + a_2*c + \dots + a_N*c$
5. $a = b \rightarrow b = a$
6. $a + \epsilon \rightarrow a$, if $\epsilon \ll a$
7. Transposition rule: $a + b = c + d \rightarrow a - c = d - b$, etc.
8. $ab / cd = a*(b/cd)$
9. $ab/cd = (ab/c)/d$

Many other transformation rules can be applied. But the above ones cover a vast majority of decompositions.

Process decomposition tree and equation parse tree cannot easily be handled in Prolog, because tree representation cannot be done easily. We can represent trees through lists, but although Prolog support list structure, we have to define the length of how far we will move using list of lists. So we had rather represented trees through the dynamic database structure of Turbo Prolog. We used three databases for this purpose. These are:

Process_tree_children_no(N,R,S,L,M)

Process_tree_child_no(N,R,S,X,I)

Process_leaf(N,R,S)

We use similar dynamic databases for the parse tree. Here N refers to the level of a node of a tree, R represents the string or name of that node, S is the number of being child of its parent node, L is the list of children of that node and X is the I-th child. As with this process tree predicates or database predicates, there are similar parse tree predicates.

After inputting the process decomposition tree and parse tree of equations, we find the same branch nodes of the process tree and have the results at the internal nodes of the parse tree. Then the process tree nodes that matches the leaves of the parse tree are inputted. Then process tree nodes matched at internal nodes of the parse tree are obtained. As has already been stated, for an additive or subtractive node, the result will be the union of the sets of matching process nodes at the child nodes of that particular node. For division and multiplication, it will be the intersection.

After having the matches at all nodes of parse tree, we seek for the result. If any node matches a single process node or nodes of the same branch, then we conclude that the equation at that root process node is the parse result at the parse node concerned. If no such node or set of nodes can be obtained, then search through operators is done using depth-first search strategy. Using an operator, the whole equation parse tree is modified and the dynamic database is also modified. Then again we search for a solution. If solution at all nodes of the process tree is found, then we terminate the search.

Now let us illustrate the method of decomposition through two case studies.

3.3.1 Case I: Water temperature problem

Now let us illustrate the system using the water temperature case mentioned earlier. We will use the process decomposition tree having a process P comprised of two subprocesses, is (initial state) and fs (final state). We will input the parse tree of the equation, $tf = (m1t1 + m2t2)/(m1 + m2)$. Matching at the leaves will then start by inputting the following correspondences:

1. $tf \rightarrow fs$
2. $m1 \rightarrow is, fs$
3. $t1 \rightarrow is$
4. $m2 \rightarrow is, fs$
5. $t2 \rightarrow is$

After matching at the leaves, matching at the internal nodes of the parse tree is done. At this stage, we get 'is' at 'm1t1', 'is' at m2t2, 'is' at 'm1t1+m2t2', 'is' at '(m1t1+m2t2)/(m1+m2)', but 'is' and 'fs' at 'm1+m2'. Since these two are not of same branch, operator search occurs.

The first operator now matches and the equation reduces to:

$$m1t1+m2t2 = tf(m1+m2)$$

Now again we do not get a single process node at 'm1+m2'. So, again we make operator search.

Now operator 3 matches and the equation again reduces to:

$$m1t1 + m2t2 = m1tf + m2tf$$

Now all the parse tree nodes match single process nodes. So the goal succeeds and we get the decomposed equations:

'is': m1t1+m2t2, m1t1, m2t2

'fs': m1tf+m2tf, m1tf, m2tf

Then a second level search attempts to drop first any of the expressions m1t1, m2t2, m1tf, m2tf using the operator $a + \epsilon \rightarrow a$, if $\epsilon \ll a$. But since no such conclusions can be made, the operator fails.

Now the transposition rule tries to succeed, and operator 6 makes transformations like:

$$m1t1 - m1tf = m2tf - m2t2$$

But this transformation cannot yield a successful decomposition, the rule fails, and the previous goal that has been succeeded is the output of the system.

3.3.2 Case II: Electrical circuit problem

We know that if only one resistor R_1 is in series with a battery having voltage E_1 with current I flowing, then from Ohm's law, we get $E_1 = IR_1$. But if E_1 and E_2 are in additive series with two resistances R_1 and R_2 , then the equation obtained from a BACON-like system for the circuit will be:

$$E_1 + E_2 = IR_1 + IR_2$$

Here we do not get the equation for the elementary interaction, i.e., Ohm's law.

So we want to decompose the equation.

We will designate the process P as being composed of two subprocesses, P_1 for battery E_1 and resistance R_1 and P_2 for battery E_2 and resistor R_2 .

Match at the leaves will yield:

1. $E_1 \rightarrow P_1$
2. $E_2 \rightarrow P_2$
3. $I \rightarrow P_1, P_2$
4. $R_1 \rightarrow P_1$
5. $R_2 \rightarrow P_2$

Now we will have both P_1 and P_2 for ' $E_1 + E_2$ ', which are not of same branch. So operator search starts. Since there are '+' sign, the operator search will seek either of its descendents is much smaller than the other. Since no such conclusion can be made, transposition rule is tried, and the equation reduces to:

$$E_1 - IR_1 = E_2 - IR_2$$

Here the left side matches with P_1 and the right side with P_2 .

Both of these matches are nothing but reflections of Ohm's law for P_1 and P_2 respectively.

Thus for a wide variety of problems the developed system successfully decomposes equations from a BACON-like system into the basic laws.

Chapter 4

Discussion and Conclusion

4.1 Evaluation of the system

The system developed successfully decomposes composite equations. Here we will illustrate two examples of the decomposition of composite equations derived from a BACON-like system.

4.1.1 Example 1

Let us first take the case of ideal gas law. We know that the system BACON can successfully discover this law [2]. But there may be a number of variations in the physical situations to describe the law. In one arrangement, the temperature of the gas may remain constant. The pressure may be constant in another experiment. So the process decomposition tree may be like figure 1.

Now the equation parse tree will be as follows:

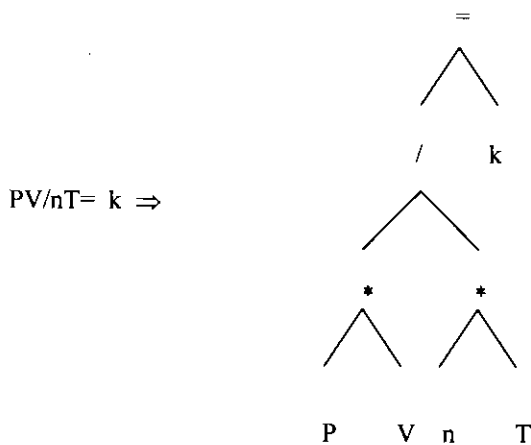


Figure 4.1: Parse tree for ideal gas law

The lists of process nodes corresponding to P, V, n, T and k are $\{p_1\}$, $\{p_1, p_2\}$, $\{p_1, p_2\}$, $\{p_2\}$ and $\{p_1, p_2\}$ respectively, where p_1 and p_2 refers to process1 and process2 respectively.

We get $\{p_1\}$ and $\{p_2\}$ respectively at the two nodes designated by '*', because these can be obtained by the intersection of the sets or lists for their children. At the node designated by '/', we will get a null set of process tree nodes. So operator search transforms the equation as: $PV = KnT$. And we get PV for process 1, where temperature is constant. This is an indication of Boyle's law.

A second level of search transforms the equation into the form

$$P = KnT/V$$

At the node for T/V, we will get 'process 2', where pressure is held constant. This is a complete indication of Charles' law.

Thus Boyle's law and Charles' law can be deduced from the ideal gas law.

4.1.2 Example 2

Consider the case of Black's experiment. Let us take two beakers containing m_1 and m_2 amount of water at temperatures t_1 and t_2 respectively. The water in the two beakers are mixed and waited a bit until, at equilibrium, the final temperature be t_f . Now the process decomposition tree for this experiment will be:

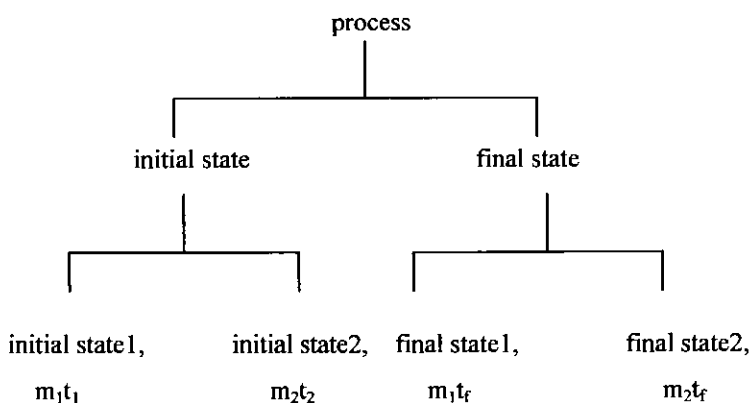


Figure 4.2: Process decomposition tree for Black's experiment

The corresponding equation parse tree will be:

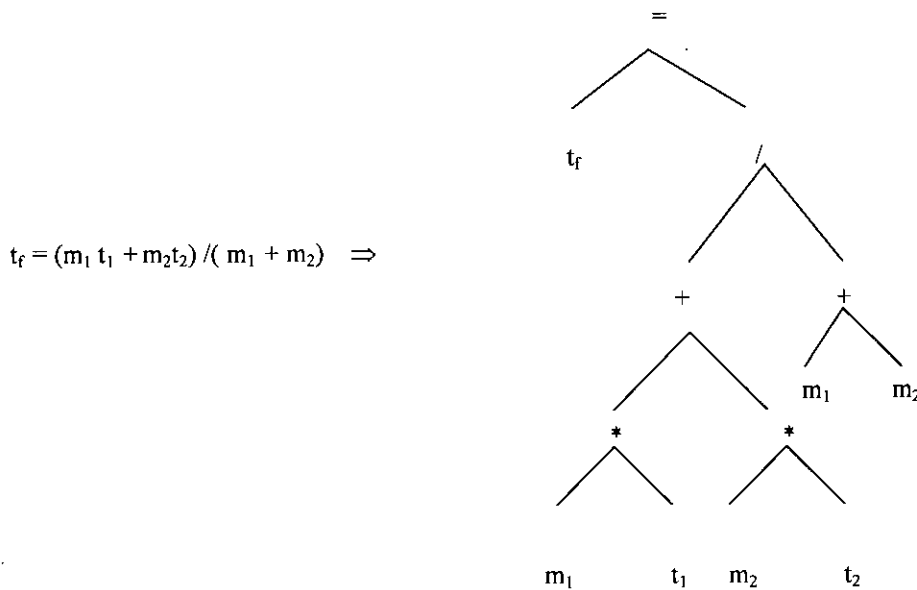


Figure 4.3: Parse tree for Black's law

Now {initial state} successfully matches up to $(m_1 t_1 + m_2 t_2)$. But the match does not reach the root. To reach the root, we should use the operator : $a = b/c \rightarrow ac = b$. Then, obviously, {final state} will match $(m_1 + m_2) t_f$, yielding the basic laws of thermal equilibrium.

4.2 Further Development

The present program incorporates a large number of mathematical operators to guide the search. Using the operator selection, we can have a huge database of relations. Still there are situations where extensions to the present system can be made. These are outlined here.

There may be discrepancies in the interpretation of processes and as such the process decomposition tree may be faulty. A section of subprocesses in the experiment may be overlooked or an extraneous portion of an experiment that has no significant impact can be included in the process interpretation. So the system may not successfully deduce the basic laws. In these cases, a system can be developed which will yield good results in such situations.

78587



In developing the system, we use depth-first search with backtracking. Heuristic searches can be incorporated to reach the goal at a minimal cost. Heuristic algorithms like A*, AO* or steepest-ascent hill-climbing can be used to guide the search to obtain an optimal solution.

We can take advantage of the finite differences to step forward to the components of differential equations. Any expression of the form $k(x_1 - x_2)$, where x_1 and x_2 are two states relating to k , can be replaced by a finite difference $k \Delta x$. When we take into account the case of infinitesimal differences, it becomes $k dx$. When such infinitesimal differences are encountered, differential equations may arise.

We have represented the process decomposition tree and the equation parse tree through the dynamic database structure of Turbo Prolog. Use of this database and deletion and at the same time update of the same during equation transformation require a large memory space, especially large stack area. Often it so occurs that stack overflows and the program can not run. So computers having high memory capacity happen to be required. This could be avoided, if some other representation except the dynamic database could be implemented in the system.

Finally the decomposition process can be used in scientific model-building process.

4.3 Conclusion

The number of AI discovery systems has been growing considerably since the last decade. These systems cover many areas of human discovery activity and are becoming useful as everyday tools for scientists, providing labor-intensive, systematic and unbiased help. Two discovery processes BACON and FAHRENHEIT play the most vital role in empirical discovery of scientific equations. Equations derived from such systems are used to deduce the basic laws in the present program.

Equation transformations and their interpretations form the main search space. There is a system called GALILEO [29], which also decompose composite equations to obtain the elementary interactions, but we implement the system differently and add some useful operators to guide the search. We use a database model for the implementation of the process tree and the equation parse tree. A trajectory is formed from an initial state, i.e, equation in the form discovered by BACON-like systems to a goal state which is an equivalent equation in interpretable form.

References

- [1] Cheeseman, P., Self, M., Kelly, J., Taylor, W., Freeman, D. & Stutz, J. (1988). Bayesian Classification, In: Proceedings of AAAI-88.
- [2] Falkenhainer, B. C. & Michalski, I. S. (1986), Integrating quantitative and qualitative discovery : the ABACUS system. *Machine Learning* 1, 367 - 401.
- [3] Fischer, D. H. (1987). Knowledge acquisition via incremental conceptual clustering, *Machine Learning*, 2, 139 - 172.
- [4] Forbus, K. D. (1984). *Qualitative Process Theory: Qualitative Reasoning about Physical Systems*. Cambridge, MA; MIT Press.
- [5] Gerwin, D. G. (1976). Information processing, data inference, and scientific generalization, *Behavioral Science*, 19, 314 - 325.
- [6] Jankowski, A. & Zytow, J. M. (1988). A methodology of multisearch systems. In: Z. Ras & L. Saita (eds.), *Methodologies for intelligent systems* (vol. 3). New York : North Holland.
- [7] Knight, K. and Rich, E. (1991), *Artificial Intelligence*, 2nd edition, Mc-Graw Hill Book Company, New York.
- [8] Koehn, B. & Zytow, J. M. (1986). Experimenting and Theorizing in Theory formation. *Proceedings of the international symposium on methodologies for intelligent systems* (pp.296-307). ACM-SIGART Press.
- [9] Kulkarni, D. & Simon, A. (1987), *The processes of Scientific Discovery: the Strategy of Experimentation*. Carnegie- Mellon University.
- [10] Langley, P. (1981). Data-driven discovery of physical laws. *Cognitive Science*, 5, 31 – 54.

- [11] Langley, P. W. & Zytkow, J. M. (1989). Data-driven approaches to empirical discovery, *Artificial Intelligence*, 40, 283 - 314.
- [12] Langley, P., Bradshaw, G. L., Simon, A. and Zytkow, J. M. (1987), *Scientific discovery: Computational Explorations of the Creative Processes*. Cambridge, MA: MIT press.
- [13] Lenat, D. (1976). AM: An AI approach to discovery in mathematics as heuristic search, Ph. D. Thesis, Computer Science Department, Stanford University, Stanford, CA.
- [14] Lenat, D. (1983). EURISKO: A program that learns new heuristics and domain concepts, *Artificial Intelligence*, 21, 61 - 98.
- [15] Lenat, D. and Brown, J., (1984). Why AM and EURISKO appear to work, *Artificial Intelligence*, 23, 269 - 294.
- [16] Lewenstam, A. & Zytkow, J. M. (1989). Model-based science of ion-selective electrodes. In: E. Pungor (Ed.), *Ion-selective Electrodes (Vol. 5)*. Oxford, England: Pergamon Press.
- [17] Moulet, M. (1992). ARC.2: Linear regression in ABACUS, In: Zytkow, J. M. (ed.), *Proceedings of ML-92 workshop on machine discovery*, Aberdeen, UK, July 4, 137 - 146.
- [18] Newell, A. & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.
- [19] Nordhausen, B. and Langley, P. (1993). An integrated framework for empirical discovery, *Machine Learning*, 12, 17 - 47.
- [20] Nordhausen, B. & Langley, P. (1990). A robust approach to numeric discovery. *Proceedings of the Seventh International Conference on Machine Learning*, Morgan Kaufman Publishers, Inc., Palo Alto, CA, 411 - 418.
- [21] Piatetsky - Shapiro, G. & Frawley, W. (eds.)(1991). *Knowledge discovery in databases*, Menlo Park, CA: AAAI Press.

- [22] Schaffer, C. (1993). Bivariate scientific function finding in a sampled, real-data testbed, *Machine Learning*, 12, 167 - 183.
- [23] Wu, Y. and Wang, S. (1989). Discovering knowledge from observational data. In: Piatetski - Shapiro, G. (ed.), *Knowledge discovery in databases, IJCAI - 89 Workshop Proceedings*, Detroit, MI, 369 - 377.
- [24] Zembowicz, R. & Zytkow, J. M. (1992). Discovery of equations: experimental evaluation of convergence, *Proceedings of 10th national conference on Artificial Intelligence*, the AAAI Press, 70 - 75.
- [25] Zembowicz, R. & Zytkow, J. M. (1996). From Contingency Tables to various forms of knowledge in databases. In: *Advance: in Knowledge Discovery and Data Mining*. In: U. Fayyad, G. Piatetsky - Shapiro, P. Smyth and R. Uthurusamy (eds.), AAAI Press.
- [26] Zytkow, J. M. (1993). Introduction: cognitive autonomy in machine discovery, *Machine Learning*, 12, 7 - 16, Kluwer Academic Publishers, Boston.
- [27] Zytkow, J. M. & Jankowski, A. (1989). Hierarchical control and heuristics in multisearch systems. In Z. Ras (Ed.), *Methodologies for intelligent systems (vol. 4)*. New York.
- [28] Zytkow, J. M. & Zhu, Jieming (1991). Automated empirical discovery in a numerical space. *Proceedings of the Third Annual Chinese Machine Learning Workshop*. Harbin, Peoples Republic of China.
- [29] Zytkow, J. M. (1990). Deriving law through analysis of processes and equations. In: Langley, P. & Shragar, J. (eds.). *Computational models of discovery and theory formation*. Morgan Kaufman Publishers Inc., San Mateo, CA, 129 - 156.
- [30] Zytkow, J. M. (1996). Automated discovery of empirical laws. *Fundamenta Informaticae*, 27, IOS press, 299 - 318.
- [31] Zytkow, J. M. (1997). Creating a discoverer: automatic knowledge seeking agent. Zytkow, J. M. (ed.). Kluwer Academic Publishers.

- [32] Zytkow, J. M., Zhu, J. & Hussam, A. (1990). Automated discovery in a chemistry laboratory, Proceedings of the AAAI - 90, AAAI Press, 889 - 894.
- [33] Zytkow, J. M., & Zembowicz, R. (1993). Database exploration in search of regularities, Journal of Intelligent Information Systems, 2, 39 - 81.

APPENDIX
PROGRAM LISTING

```
/******
```

```
THIS PROGRAM IS WRITTEN BY  
MOHAMMAD ISMAT KADIR.
```

```
*****/
```

```
/*          DECLARATION SECTION          */
```

```
domains
```

```
root, child, rootp, head, element, x1, x, other=string  
n, m, i, nn, mm, mp, j, jj, k, kk, il=integer  
list, new_list, listp, tail, list2, list3, list1, listb,  
new, set1, set, set2, newtail, lists? ash, rem, rem1=string*
```

```
predicates
```

```
get_process_tree  
get_child(integer, integer, string, integer, list)  
test1(integer, integer, string, integer, string, list)  
  
get_parse_tree  
get_childp(integer, integer, string, integer, integer, list)  
test2(integer, integer, string, integer, integer, string, list)  
append_list(string, list, list)  
n_element(list, integer, string)  
  
rept(list, integer, integer, integer)  
reptp(list, integer, integer, integer, integer)  
operator_search  
op1  
op2  
op3  
op4  
op5  
op6  
op7  
op8  
op9  
construct_slash_list(integer, list)  
loop(integer, integer, integer, integer, integer, list, string)  
construct_star_list(integer, list)  
looping(integer, integer, integer, integer, integer, list, string)  
looping2(integer, integer, integer, integer, integer, list, string)  
loop_neg(integer, integer, integer, integer, integer, list, string)  
looping_neg(integer, integer, integer, integer, integer, list, string)  
looping_neg2(integer, integer, integer, integer, integer, list, string)
```

```

include_child(integer, string, integer, integer, integer, integer, list, integer,
integer)
equal_check(integer, integer, integer)
pull(string, list, list)
retract_next_child(integer, list, integer, integer, integer)
delete_next_child(integer, string, integer)
compare1(list, integer, integer, integer, string, integer)
compare2(list, integer, string, integer, integer, integer, string, integer)
response(string)
get_parse_result
parse_tree_result(integer, string, integer, integer)
store_parse_result(integer, integer, integer, string, string)

find_result(integer, list, string, integer, integer, integer, integer, string)
match
matching_leaf_list(integer, string, integer, integer, list)
matching_nodes_list(integer, string, integer, integer, list, list)
match_internal(integer, string, integer, integer, integer, integer)
replit(integer, string, integer, integer, integer, integer, list)
internal_match(integer, string, integer, integer, integer)
retract_children(integer, string, integer, integer)
retract_child(integer, string, integer, integer)
retract_result
deduct(list, list, list)
add(string, list, list)
retract_match
assertion
accumulation
accumulate(integer, string, integer, integer, list, integer, integer)
speak(integer, string, integer, integer, integer, integer)
length(list, integer)
union(list, list, list)
intersect(list, list, list)
append(list, list, list)
replace_with(list, string, integer, integer, list, list)
member(string, list)

correcting_child(integer, integer, string, integer, integer, integer, list, integer, integer)
correcting_child1(integer, integer, string, integer, list, integer, integer)
investigate_node1(integer, integer, string, string, string)
investigate_node2(integer, integer, string, string, string)
investigate_node11(integer, integer, string, string, string)
investigate_node21(integer, integer, string, string, string)
investigate_node12(integer, integer, string, string, string)
investigate_node22(integer, integer, string, string, string)
investigate_node13(integer, integer, string, string, string)
investigate_node23(integer, integer, string, string, string)
next_investigate(integer, integer, string, integer, integer, integer)
next_investigatel(integer, integer, string, integer, integer)
check(string)
equal_list(list, list)
whether_same_process_branch
same_branch(list, integer, integer)
find_root(list, string)

```



```

same_branch_check(string)
next_check(string,list,list,integer,integer)
checkA(integer,string,integer,integer,integer,integer)

judge(string,integer,integer,integer,integer,string,integer,integer,integer,integer)
go

database

process_leaf(integer,string,integer)
process_tree_children_no(integer,string,integer,list,integer)
process_tree_child_no(integer,string,integer,string,integer)
same_process_branch(string,list)
parse_leaf(integer,string,integer,integer)
parse_tree_children_no(integer,string,integer,integer,list,integer)
parse_tree_child_no(integer,string,integer,integer,string,integer)
parse_result(integer,integer,integer,string,string)
matching_nodes(integer,string,integer,integer,list)
matched_leaf(string,list)

goal
go.

/*      PROCESS TREE INPUT SECTION      */

clauses

go:-
    get_process_tree,
    whether_same_process_branch,
    get_parse_tree,
    get_parse_result,
    match,
    accumulation,
    assertion,
    operator_search,
    retract_result,
    retract_match,
    get_parse_result,
    match,
    accumulation,
    assertion.

get_process_tree:-
    makewindow(1,7,7,"PROCESS TREE INPUT",4,5,20,50),

```

```

        write("Enter the root of process tree:"),
        readln(Root),
        write("Enter the children of the process tree root,
'",Root,"':"),
        nl,get_child(1,0,Root,1, []).

get_child(N,M,Root,S,List):-
    readln(Child),
    test1(N,M,Root,S,Child,List).

test1(N,M,Root,S,Child,List):-
    Child<>"",!,
    MM=M+1,
    assertz(process_tree_child_no(N,Root,S,Child,MM)),
    append_list(Child,List,New_list),
    get_child(N,MM,Root,S,New_list).
test1(N,M,Root,S,Child,List):-
    M>0,!,
    assertz(process_tree_children_no(N,Root,S,List,M)),
    NN=N+1,
    MM=M+1,
    rept(List,NN,1,MM).
test1(N,M,Root,S,Child,List):-
    !,
    asserta(process_leaf(N,Root,S)).
rept(List,NN,MM,MM).
rept(List,NN,I,MM):-
    n_element(List,I,X),
    write("Enter the children of the process tree root,
'",X,"':"),
    nl,get_child(NN,0,X,I, []),
    II=I+1,
    rept(List,NN,II,MM).

append_list(Element,[],[Element]).
append_list(Element,[Head|Tail],[Head|Newtail):-
    append_list(Element,Tail,Newtail).

n_element([Head|_],1,X1):-
    !,
    X1=Head.
n_element([_Head|Tail],N,X1):-
    NN=N-1,
    n_element(Tail,NN,X1).

/*          PARSE TREE INPUT SECTION          */

get_parse_tree:-

```

```

makewindow(2,7,7," PARSE TREE INPUT
SECTION",3,5,18,50),
write(" Enter the root of parse tree :"),
readln(Root),
write("Enter the children of the parse tree
root, '',Root, '' :"),
nl,get_childp(1,0,Root,1,1, []).

get_childp(N,M,Root,S,K,List):-
readln(Child),
test2(N,M,Root,S,K,Child,List).

test2(N,M,Root,S,K,Child,List):-

Child<>"",!,
MM=M+1,
assertz(parse_tree_child_no(N,Root,S,K,Child,MM)),
append_list(Child,List,New_list),
get_childp(N,MM,Root,S,K,New_list).

test2(N,M,Root,S,K,Child,List):-
M>0,!,
assertz(parse_tree_children_no(N,Root,S,K,List,M)),
NN=N+1,
MM=M+1,
reptp(List,S,NN,1,MM).
test2(N,M,Root,S,K,Child,List):-
!,
asserta(parse_leaf(N,Root,S,K)).
reptp(L,K,N,M,M):-!.
reptp(L,K,N,I,M):-
n_element(L,I,X),
write("Enter the children of the parse tree
root, '',X, '' :"),
nl,get_childp(N,0,X,I,K, []),
II=I+1,
reptp(L,K,N,II,M).

/***** THIS SECTION USES THE OPERATOR SEARCH
*****/

```

```

operator_search:-
op1.
operator_search:-
op2.
operator_search:-
op3.
operator_search:-
op4.
operator_search:-
op5.
operator_search:-
op6.
operator_search:-
op7.
operator_search:-

```

```

        op8.
operator_search:-
        op9.
operator_search.

```

/**** OP1 makes use of the operation, a=b/c gives ac=b.****/

op1:-

```

parse_tree_children_no(N,"=",S,1,List,2),
parse_tree_child_no(N,"=",S,1,A,2),
parse_tree_child_no(N,"=",S,1,"/",1),
NN=N+1,
parse_tree_child_no(NN,"/",1,S,C,2),
parse_tree_child_no(NN,"/",1,S,B,1),
N1=NN+1,
!,
retract_child(N,"=",S,1),
retract_children(N,"=",S,1),
retract_child(NN,"/",1,S),
retract_children(NN,"/",1,S),
asserta(parse_tree_child_no(N,"=",S,1,B,1)),
asserta(parse_tree_child_no(N,"=",S,1,"*",2)),
asserta(parse_tree_children_no(N,"=",S,1,[B,"*"],2)),
asserta(parse_tree_children_no(NN,"*",2,1,[A,C],2)),
asserta(parse_tree_child_no(NN,"*",2,1,A,1)),
asserta(parse_tree_child_no(NN,"*",2,1,C,2)),
investigate_nodel(NN,N1,A,B,C).

```

op1:-

```

parse_tree_children_no(N,"=",S,1,List,2),
parse_tree_child_no(N,"=",S,1,A,1),
parse_tree_child_no(N,"=",S,1,"/",2),
NN=N+1,
parse_tree_child_no(NN,"/",2,S,C,2),
parse_tree_child_no(NN,"/",2,S,B,1),
N1=NN+1,
retract_child(N,"=",S,1),
retract_children(N,"=",S,1),
retract_child(NN,"/",2,S),
retract_children(NN,"/",2,S),
asserta(parse_tree_child_no(N,"=",S,1,B,1)),
asserta(parse_tree_child_no(N,"=",S,1,"*",2)),
asserta(parse_tree_children_no(N,"=",S,1,[B,"*"],2)),
asserta(parse_tree_children_no(NN,"*",2,1,[A,C],2)),
asserta(parse_tree_child_no(NN,"*",2,1,A,1)),
asserta(parse_tree_child_no(NN,"*",2,1,C,2)),
investigate_node2(NN,N1,A,B,C).

```

```

correcting_child(NN,N1,A,J,J1,JJ,A1,K,K):-!,
K2=K-1,

```

```

asserta(parse_tree_children_no(N1,A,J,J1,A1,K2)).

```

```

correcting_child(NN,N1,A,J,J1,JJ,A1,K,I1):-

```

```

n_element(A1,I1,H),K2=K-1,
asserta(parse_tree_child_no(N1,A,J,J1,H,I1)),
next_investigate(NN,N1,H,I1,J,JJ),
II=I1+1,
correcting_child(NN,N1,A,J,J1,JJ,A1,K,II).

```

```

next_investigate(NN,N1,H,I1,J,JJ):-
    N=NN+1,N2=N1+1,
    parse_tree_children_no(N,H,I1,JJ,H1,K2),!,
    retract_children(N,H,I1,JJ),
    retract_child(N,H,I1,JJ),
    K=K2+1,
    correcting_child(N,N2,H,I1,J,I1,H1,K,1).
next_investigate(NN,N1,H,I1,J,JJ):-
    N=NN+1,N2=N1+1,
    parse_leaf(N,H,I1,JJ),!,
    retract(parse_leaf(N,H,I1,JJ)),
    asserta(parse_leaf(N2,H,I1,J)).

investigate_node1(NN,N1,A,B,C):-
    investigate_node11(NN,N1,A,B,C),
    investigate_node12(NN,N1,A,B,C),
    investigate_node13(NN,N1,A,B,C).

investigate_node11(NN,N1,A,B,C):-
    parse_leaf(NN,A,2,1),!,
    retract(parse_leaf(NN,A,2,1)),
    asserta(parse_leaf(N1,A,1,2)).

investigate_node11(NN,N1,A,B,C):-
    parse_tree_children_no(NN,A,2,1,A1,K2),
    retract_child(NN,A,2,1),
    retract_children(NN,A,2,1),K=K2+1,
    correcting_child(NN,N1,A,1,2,2,A1,K,1).

investigate_node12(NN,N1,A,B,C):-
    parse_leaf(N1,B,1,1),!,
    retract(parse_leaf(N1,B,1,1)),
    asserta(parse_leaf(NN,B,1,1)).

investigate_node12(NN,N1,A,B,C):-
    parse_tree_children_no(N1,B,1,1,B1,K1),
    retract_child(N1,B,1,1),
    retract_children(N1,B,1,1),K=K1+1,
    correcting_child(N1,NN,B,1,1,1,B1,K,1).

investigate_node13(NN,N1,A,B,C):-
    parse_leaf(N1,C,2,1),!,
    retract(parse_leaf(N1,C,2,1)),
    asserta(parse_leaf(N1,C,2,2)).

investigate_node13(NN,N1,A,B,C):-
    parse_tree_children_no(N1,C,2,1,C1,K3),
    retract_child(N1,C,2,1),
    retract_children(N1,C,2,1),K=K3+1,
    correcting_child(N1,N1,C,2,2,2,C1,K,1).

investigate_node2(NN,N1,A,B,C):-

```

```

investigate_node21(NN,N1,A,B,C),
investigate_node22(NN,N1,A,B,C),
investigate_node23(NN,N1,A,B,C).

```

```

investigate_node21(NN,N1,A,B,C):-
    parse_leaf(NN,A,1,1),!,
    retract(parse_leaf(NN,A,1,1)),
    asserta(parse_leaf(N1,A,1,2)).

```

```

investigate_node21(NN,N1,A,B,C):-
    parse_tree_children_no(NN,A,1,1,A1,K2),
    retract_child(NN,A,1,1),
    retract_children(NN,A,1,1),K=K2+1,
    correcting_child(NN,N1,A,1,2,1,A1,K,1).

```

```

investigate_node22(NN,N1,A,B,C):-
    parse_leaf(N1,B,1,2),!,
    retract(parse_leaf(N1,B,1,2)),
    asserta(parse_leaf(NN,B,1,2)).

```

```

investigate_node22(NN,N1,A,B,C):-
    parse_tree_children_no(N1,B,1,2,B1,K1),
    retract_child(N1,B,1,2),
    retract_children(N1,B,1,2),K=K1+1,
    correcting_child(N1,NN,B,1,1,1,B1,K,1).

```

```

investigate_node23(NN,N1,A,B,C):-
    parse_leaf(N1,C,2,2),!,
    retract(parse_leaf(N1,C,2,2)),
    asserta(parse_leaf(N1,C,2,2)).

```

```

investigate_node23(NN,N1,A,B,C):-
    parse_tree_children_no(N1,C,2,2,C1,K3),
    retract_child(N1,C,2,2),
    retract_children(N1,C,2,2),K=K3+1,
    correcting_child(N1,N1,C,2,2,2,C1,K,1).

```

```

/***** op2 makes use of the operation
*****
***** (a+b+c+.....)/c gives
a/c+b/c+c/c+..... *****/

```

op2:-

```

    parse_tree_children_no(N,"/",S,T,["+",C],2),
    NN=N+1,
    parse_tree_children_no(NN,"+",1,S,List,M),!,
    M1=M+1,
    construct_slash_list(M,Listslash),NM=N-1,
    parse_tree_child_no(NM,R,K,U,"/",I),
    retract(parse_tree_child_no(NM,R,K,U,"/",I)),
    asserta(parse_tree_child_no(NM,R,K,U,"+",I)),
    parse_tree_children_no(NM,R,K,U,LL,MM),
    retract(parse_tree_children_no(NM,R,K,U,LL,MM)),
    replace_with(LL,"+",I,1,[],KK),
    asserta(parse_tree_children_no(NM,R,K,U,KK,MM)),
    asserta(parse_tree_children_no(N,"+",S,T,Listslash,M)),
    loop(1,M1,N,S,T,Listslash,C),

```

```

retract_children(N, "/", S, T),
retract_child(N, "/", S, T),
retract_child(NN, C, 2, S),
retract_child(NN, "+", 1, S),
retract_children(NN, "+", 1, S).

```

op2:-

```

parse_tree_children_no(N, "/", S, T, ["-", C], 2),
NN=N+1,
parse_tree_children_no(NN, "-", 1, S, List, M), !,
M1=M+1,
construct_slash_list(M, Listslash), NM=N-1,
parse_tree_child_no(NM, R, K, U, "/", I),
retract(parse_tree_child_no(NM, R, K, U, "/", I)),
asserta(parse_tree_child_no(NM, R, K, U, "-", I)),
parse_tree_children_no(NM, R, K, U, LL, MM),
retract(parse_tree_children_no(NM, R, K, U, LL, MM)),
replace_with(LL, "-", I, 1, [], KK),
asserta(parse_tree_children_no(NM, R, K, U, KK, MM)),
asserta(parse_tree_children_no(N, "-", S, T, Listslash, M)),
loop_neg(1, M1, N, S, T, Listslash, C),
retract_children(N, "/", S, T),
retract_child(N, "/", S, T),
retract_child(NN, C, 2, S),
retract_child(NN, "-", 1, S),
retract_children(NN, "-", 1, S).

```

```

construct_slash_list(0, []) :-!.
construct_slash_list(M, ["/"|List]) :-
    MM=M-1,
    construct_slash_list(MM, List).

```

```

loop(M, M, N, L, T, Listslash, C) :-!.
loop(I, M, N, L, T, Listslash, C) :-
    NN=N+1,
    parse_tree_child_no(NN, "+", 1, L, X, I),
    asserta(parse_tree_child_no(N, "+", L, T, "/", I)),
    asserta(parse_tree_child_no(NN, "/", I, L, X, 1)),
    asserta(parse_tree_child_no(NN, "/", I, L, C, 2)),
    asserta(parse_tree_children_no(NN, "/", I, L, [X, C], 2)),
    N1=NN+1, M1=M-1,
    judge(X, I, 1, M1, C, L, 2, NN, N1),
    II=I+1,
    loop(II, M, N, L, T, Listslash, C).

```

```

loop_neg(M, M, N, L, T, Listslash, C) :-!.
loop_neg(I, M, N, L, T, Listslash, C) :-
    NN=N+1,
    parse_tree_child_no(NN, "-", 1, L, X, I),
    asserta(parse_tree_child_no(N, "-", L, T, "/", I)),
    asserta(parse_tree_child_no(NN, "/", I, L, X, 1)),
    asserta(parse_tree_child_no(NN, "/", I, L, C, 2)),
    asserta(parse_tree_children_no(NN, "/", I, L, [X, C], 2)),
    N1=NN+1, M1=M-1,

```

```

judge(X,I,1,M1,C,L,2,NN,N1),
II=I+1,
loop_neg(II,M,N,L,T,Listslash,C).

```

```

/***** This operator makes use of the relation*****/
/*****(a+b+.....)*c = a*c+b*c+.....*****/

```

op3:-

```

parse_tree_children_no(N,"*",S,T,["+",C],2),
NN=N+1,
parse_tree_children_no(NN,"+",1,S,List,M),!,
M1=M+1,
construct_star_list(M,Liststar),NM=N-1,
parse_tree_child_no(NM,R,K,U,"*",I),
retract(parse_tree_child_no(NM,R,K,U,"*",I)),
asserta(parse_tree_child_no(NM,R,K,U,"+",I)),
parse_tree_children_no(NM,R,K,U,LL,MM),
retract(parse_tree_children_no(NM,R,K,U,LL,MM)),
replace_with(LL,"+",I,1,[],KK),
asserta(parse_tree_children_no(NM,R,K,U,KK,MM)),
asserta(parse_tree_children_no(N,"+",S,T,Liststar,M)),
looping(1,M1,N,S,T,Liststar,C),
retract_children(N,"*",S,T),
retract_child(N,"*",S,T),
retract_child(NN,C,2,S),
retract_child(NN,"+",1,S),
retract_children(NN,"+",1,S).

```

```

/***** This operator makes use of the relation*****/
/*****(a-b-.....)*c = a*c-b*c-.....*****/

```

op3:-

```

parse_tree_children_no(N,"*",S,T,["-",C],2),
NN=N+1,
parse_tree_children_no(NN,"-",1,S,List,M),!,
M1=M+1,
construct_star_list(M,Liststar),NM=N-1,
parse_tree_child_no(NM,R,K,U,"*",I),
retract(parse_tree_child_no(NM,R,K,U,"*",I)),
asserta(parse_tree_child_no(NM,R,K,U,"-",I)),
parse_tree_children_no(NM,R,K,U,LL,MM),
retract(parse_tree_children_no(NM,R,K,U,LL,MM)),
replace_with(LL,"-",I,1,[],KK),
asserta(parse_tree_children_no(NM,R,K,U,KK,MM)),
asserta(parse_tree_children_no(N,"-",S,T,Liststar,M)),
looping_neg(1,M1,N,S,T,Liststar,C),
retract_children(N,"*",S,T),
retract_child(N,"*",S,T),
retract_child(NN,C,2,S),
retract_child(NN,"-",1,S),
retract_children(NN,"-",1,S).

```



```

/***** This operator makes use of the relation*****/
/***** c*(a+b+.....) = a*c+b*c+.....*****/

```

op3:-

```

parse_tree_children_no(N,"*",S,T,[C,"+"],2),
NN=N+1,
parse_tree_children_no(NN,"+",2,S,List,M),!,
M1=M+1,
construct_star_list(M,Liststar),NM=N-1,
parse_tree_child_no(NM,R,K,U,"*",I),
retract(parse_tree_child_no(NM,R,K,U,"*",I)),
asserta(parse_tree_child_no(NM,R,K,U,"+",I)),
parse_tree_children_no(NM,R,K,U,LL,MM),
retract(parse_tree_child_no(NM,R,K,U,LL,MM)),
replace_with(LL,"+",I,1,[],KK),
asserta(parse_tree_children_no(NM,R,K,U,KK,MM)),
asserta(parse_tree_children_no(N,"+",S,T,Liststar,M)),
looping2(1,M1,N,S,T,Liststar,C),
retract_children(N,"*",S,T),
retract_child(N,"*",S,T),
retract_child(NN,C,1,S),
retract_child(NN,"+",2,S),
retract_children(NN,"+",2,S).

```

```

/***** This operator makes use of the relation*****/
/***** c*(a-b-.....) = a*c-b*c-.....*****/

```

op3:-

```

parse_tree_children_no(N,"*",S,T,[C,"-"],2),
NN=N+1,
parse_tree_children_no(NN,"-",2,S,List,M),!,
M1=M+1,
construct_star_list(M,Liststar),NM=N-1,
parse_tree_child_no(NM,R,K,U,"*",I),
retract(parse_tree_child_no(NM,R,K,U,"*",I)),
asserta(parse_tree_child_no(NM,R,K,U,"-",I)),
parse_tree_children_no(NM,R,K,U,LL,MM),
retract(parse_tree_children_no(NM,R,K,U,LL,MM)),
replace_with(LL,"-",I,1,[],KK),
asserta(parse_tree_children_no(NM,R,K,U,KK,MM)),
asserta(parse_tree_children_no(N,"-",S,T,Liststar,M)),
looping_neg2(1,M1,N,S,T,Liststar,C),
retract_children(N,"*",S,T),
retract_child(N,"*",S,T),
retract_child(NN,C,1,S),
retract_child(NN,"-",2,S),
retract_children(NN,"-",2,S).

```

```

construct_star_list(0,[]):-!.
construct_star_list(M,["*"|List]):-
MM=M-1,
construct_star_list(MM,List).

```

```

looping(M,M,N,L,T,Liststar,C):-!.

```

```

looping(I,M,N,L,T,Liststar,C):-
    NN=N+1,
    parse_tree_child_no(NN,"+",1,L,X,I),!,
    asserta(parse_tree_child_no(N,"+",L,T,"*",I)),
    asserta(parse_tree_child_no(NN,"*",I,L,X,1)),
    asserta(parse_tree_child_no(NN,"*",I,L,C,2)),
    asserta(parse_tree_children_no(NN,"*",I,L,[X,C],2)),
    N1=NN+1,M1=M-1,
    judge(X,I,1,M1,C,L,2,NN,N1),
    II=I+1,
    looping(II,M,N,L,T,Liststar,C).

```

```

looping2(M,M,N,L,T,Liststar,C):-!.

```

```

looping2(I,M,N,L,T,Liststar,C):-
    NN=N+1,
    parse_tree_child_no(NN,"+",2,L,X,I),
    asserta(parse_tree_child_no(N,"+",L,T,"*",I)),
    asserta(parse_tree_child_no(NN,"*",I,L,X,1)),
    asserta(parse_tree_child_no(NN,"*",I,L,C,2)),
    asserta(parse_tree_children_no(NN,"*",I,L,[X,C],2)),
    N1=NN+1,M1=M-1,
    judge(X,I,2,M1,C,L,1,NN,N1),
    II=I+1,
    looping2(II,M,N,L,T,Liststar,C).

```

```

looping_neg(M,M,N,L,T,Liststar,C):-!.

```

```

looping_neg(I,M,N,L,T,Liststar,C):-
    NN=N+1,
    parse_tree_child_no(NN,"-",1,L,X,I),
    asserta(parse_tree_child_no(N,"-",L,T,"*",I)),
    asserta(parse_tree_child_no(NN,"*",I,L,X,1)),
    asserta(parse_tree_child_no(NN,"*",I,L,C,2)),
    asserta(parse_tree_children_no(NN,"*",I,L,[X,C],2)),
    N1=NN+1,M1=M-1,
    judge(X,I,1,M1,C,L,2,NN,N1),
    II=I+1,
    looping_neg(II,M,N,L,T,Liststar,C).

```

```

looping_neg2(M,M,N,L,T,Liststar,C):-!.

```

```

looping_neg2(I,M,N,L,T,Liststar,C):-
    NN=N+1,
    parse_tree_child_no(NN,"-",2,L,X,I),
    asserta(parse_tree_child_no(N,"-",L,T,"*",I)),
    asserta(parse_tree_child_no(NN,"*",I,L,X,1)),
    asserta(parse_tree_child_no(NN,"*",I,L,C,2)),
    asserta(parse_tree_children_no(NN,"*",I,L,[X,C],2)),
    N1=NN+1,M1=M-1,
    judge(X,I,2,M1,C,L,1,NN,N1),
    II=I+1,
    looping_neg2(II,M,N,L,T,Liststar,C).

```

```

judge(X,M,J,M,C,L,J1,NN,N1):-
    parse_leaf(NN,C,J1,L),!,
    retract(parse_leaf(NN,C,J1,L)),

```

```

asserta(parse_leaf(N1,C,2,M)),
next_investigatel(NN,NN,X,M,J).

judge(X,I,J,M,C,L,J1,NN,N1):-
    parse_leaf(NN,C,J1,L),!,
    asserta(parse_leaf(N1,C,2,I)),
    next_investigatel(NN,NN,X,I,J).

judge(X,I,J,M,C,L,J1,NN,N1):-
    parse_tree_children_no(NN,C,J1,L,L1,K),
    retract_child(NN,C,J1,L),
    retract_children(NN,C,J1,L),K1=K+1,
    correcting_child(NN,N1,C,2,I,J1,L1,K1,1).

correcting_child1(NN,N1,A,J,A1,K,K):-!,
    K2=K-1,
    asserta(parse_tree_children_no(N1,A,1,J,A1,K2)).
correcting_child1(NN,N1,A,J,A1,K,I1):-
    n_element(A1,I1,H),K2=K-1,
    asserta(parse_tree_child_no(N1,A,1,J,H,I1)),
    next_investigate(NN,N1,H,I1,1,I1),
    I1=I1+1,
    correcting_child1(NN,N1,A,J,A1,K,I1).

next_investigatel(NN,N1,H,I1,J):-
    N=NN+1,N2=N1+1,
    parse_tree_children_no(N,H,I1,J,H1,K2),!,
    retract_children(N,H,I1,J),
    retract_child(N,H,I1,J),
    K=K2+1,
    correcting_child1(N,N2,H,I1,H1,K,1).
next_investigatel(NN,N1,H,I1,J):-
    N=NN+1,N2=N1+1,
    parse_leaf(N,H,I1,J),!,
    retract(parse_leaf(N,H,I1,J)),
    asserta(parse_leaf(N2,H,1,I1)).

replace_with(LL,A,I,I,K1,KK):-
    length(LL,I),!,append(K1,[A],KK).

replace_with(LL,A,I,I,K1,KK):-
    !,append(K1,[A],Y),
    JJ=I+1,
    replace_with(LL,A,I,JJ,Y,KK).

replace_with(LL,A,I,J,K1,KK):-
    length(LL,K),J<K,!,
    n_element(LL,J,X),
    append(K1,[X],Y),
    JJ=J+1,
    replace_with(LL,A,I,JJ,Y,KK).

replace_with(LL,A,I,J,K1,KK):-
    n_element(LL,J,X),
    append(K1,[X],KK).

```

/**** OP4 makes use of the operation, $a=b*c$ gives $a/c=b$.****/

op4:-

```
parse_tree_children_no(N,"=",S,1,List,2),
parse_tree_child_no(N,"=",S,1,A,2),
parse_tree_child_no(N,"=",S,1,"*",1),
NN=N+1,
parse_tree_child_no(NN,"*",1,1,C,2),
parse_tree_child_no(NN,"*",1,1,B,1),
N1=NN+1,
!,
retract_child(N,"=",S,1),
retract_children(N,"=",S,1),
retract_child(NN,"*",1,1),
retract_children(NN,"*",1,1),
asserta(parse_tree_child_no(N,"=",S,1,B,1)),
asserta(parse_tree_child_no(N,"=",S,1,"/",2)),
asserta(parse_tree_children_no(N,"=",S,1,[B,"/"],2)),
asserta(parse_tree_children_no(NN,"/",2,1,[A,C],2)),
asserta(parse_tree_child_no(NN,"/",2,1,A,1)),
asserta(parse_tree_child_no(NN,"/",2,1,C,2)),
investigate_node1(NN,N1,A,B,C).
```

op4:-

```
parse_tree_children_no(N,"=",S,1,List,2),
parse_tree_child_no(N,"=",S,1,A,1),
parse_tree_child_no(N,"=",S,1,"*",2),
NN=N+1,
parse_tree_child_no(NN,"*",2,1,C,2),
parse_tree_child_no(NN,"*",2,1,B,1),
N1=NN+1,
retract_child(N,"=",S,1),
retract_children(N,"=",S,1),
retract_child(NN,"*",2,1),
retract_children(NN,"*",2,1),
asserta(parse_tree_child_no(N,"=",S,1,B,1)),
asserta(parse_tree_child_no(N,"=",S,1,"/",2)),
asserta(parse_tree_children_no(N,"=",S,1,[B,"/"],2)),
asserta(parse_tree_children_no(NN,"/",2,1,[A,C],2)),
asserta(parse_tree_child_no(NN,"/",2,1,A,1)),
asserta(parse_tree_child_no(NN,"/",2,1,C,2)),
investigate_node2(NN,N1,A,B,C).
```

op5:-

/**** This operator makes use of the approximation ****/
/**** $a+x = a$, where $x \ll a$ ******/

```
parse_tree_children_no(N,"+",S,T,List,M),
M1=M+1,
compare1(List,1,M1,N,"+",S),fail.
```

```

compare1 (List, M, M, N, R, S) :-!.
compare1 (List, I, M, N, R, S) :-
    n_element (List, I, X),
    compare2 (List, 1, X, I, M, N, R, S),
    II=I+1,
    compare1 (List, II, M, N, R, S).

compare2 (List, M, X, I, M, N, R, S) :-!.
compare2 (List, J, X, I, M, N, R, S) :-
    equal_check (I, J, J1),
    n_element (List, J1, Y),
    NN=N+1,
    parse_result (NN, S, I, X, X1).
    parse_result (NN, S, J1, Y, Y1),
    write("Is      '", X1, "' of same dimension as      '", Y1, "' ? (y/n):
"),
    response (Reply),
    check (Reply),
    write("Is      '", X1, "' <<      '", Y1, "' ? (y/n):      "),
    response (Rply),
    check (Rply),
    parse_tree_child_no (N, R, S, T, Child, I),
    retract (parse_tree_child_no (N, R, S, T, Child, I)),
    retract (parse_tree_children_no (N, R, S, T, List, M)),
    pull (Child, List, ListA),
    MM=M-1,
    asserta (parse_tree_children_no (N, R, S, T, ListA, MM)),
    delete_next_child (NN, Child, I),
    JJ=J1+1,
    compare2 (List, JJ, X, I, M, N, R, S).

compare2 (List, J, X, I, M, N, R, S) :-
    JJ=J+1,
    compare2 (List, JJ, X, I, M, N, R, S) .

equal_check (J, J, J1) :-!, J1=J+1.
equal_check (I, J, J).

delete_next_child (N, Child, I) :-
    parse_tree_children_no (N, Child, I, K, X, J), !,
    retract_child (N, Child, I, K),
    retract (parse_tree_children_no (N, Child, I, K, X, J)),
    NN=N+1, JJ=J+1,
    retract_next_child (NN, X, 1, I, JJ).

delete_next_child (N, Child, I) :-
    parse_leaf (N, Child, I, K),
    retract (parse_leaf (N, Child, I, K)).

retract_next_child (N, X, J, K, J) :-!.
retract_next_child (N, X, I, K, J) :-
    n_element (X, I, Y),
    parse_leaf (N, Y, I, K),
    retract (parse_leaf (N, Y, I, K)),
    II=I+1,

```

```
retract_next_child(N,X,II,K,J).
```

```
retract_next_child(N,X,I,K,J):-  
  n_element(X,I,Y),  
  retract_child(N,Y,I,K),  
  retract_children(N,Y,I,K),  
  II=I+1,  
  retract_next_child(N,X,II,K,J).
```

```
/***** This operator makes use of the transposition law *****/  
/***** a+b=c+d gives a-d=c-b, etc. *****/
```

op6:-

```
  parse_tree_children_no(N,"=",S,T,["+", "+"],2),  
  NN=N+1,  
  parse_tree_children_no(NN,"+",1,S,[A,B],2),  
  parse_tree_children_no(NN,"+",2,S,[C,D],2),!,  
  retract(parse_tree_children_no(N,"=",S,T,["+", "+"],2)),  
  retract_child(N,"=",S,T),  
  retract_children(NN,"+",1,S),  
  retract_child(NN,"+",1,S),  
  retract_children(NN,"+",2,S),  
  retract_child(NN,"+",2,S),  
  asserta(parse_tree_children_no(N,"=",S,T,["-", "-"],2)),  
  asserta(parse_tree_child_no(N,"=",S,T,"-",1)),  
  asserta(parse_tree_child_no(N,"=",S,T,"-",2)),  
  asserta(parse_tree_children_no(NN,"-",1,S,[A,C],2)),  
  asserta(parse_tree_child_no(NN,"-",1,S,A,1)),  
  asserta(parse_tree_child_no(NN,"-",1,S,C,2)),  
  asserta(parse_tree_children_no(NN,"-",2,S,[D,B],2)),  
  asserta(parse_tree_child_no(NN,"-",2,S,D,1)),  
  asserta(parse_tree_child_no(NN,"-",2,S,B,2)),  
  N1=NN+1,  
  checkA(N1,A,1,1,1,1),  
  checkA(N1,B,2,2,1,2),  
  checkA(N1,C,1,2,2,1),  
  checkA(N1,D,2,1,2,2).
```

op6:-

```
  parse_tree_children_no(N,"=",S,T,["+", "+"],2),  
  NN=N+1,  
  parse_tree_children_no(NN,"+",1,S,[A,B],2),  
  parse_tree_children_no(NN,"+",2,S,[C,D],2),!,  
  retract(parse_tree_children_no(N,"=",S,T,["+", "+"],2)),  
  retract_child(N,"=",S,T),  
  retract_children(NN,"+",1,S),  
  retract_child(NN,"+",1,S),  
  retract_children(NN,"+",2,S),  
  retract_child(NN,"+",2,S),  
  asserta(parse_tree_children_no(N,"=",S,T,["-", "-"],2)),  
  asserta(parse_tree_child_no(N,"=",S,T,"-",1)),
```

```

asserta(parse_tree_child_no( N, "=", S, T, "-", 2)),
asserta(parse_tree_children_no(NN, "-", 1, S, [A, D], 2)),
asserta(parse_tree_child_no(NN, "-", 1, S, A, 1)),
asserta(parse_tree_child_no( NN, "-", 1, S, D, 2)),
asserta(parse_tree_children_no(NN, "-", 2, S, [C, B], 2)),
asserta(parse_tree_child_no(NN, "-", 2, S, C, 1)),
asserta(parse_tree_child_no(NN, "-", 2, S, B, 2)),
N1=NN+1,
checkA(N1, A, 1, 1, 1, 1),
checkA(N1, B, 2, 2, 1, 2),
checkA(N1, C, 1, 1, 2, 2),
checkA(N1, D, 2, 2, 2, 1).

```

op6:-

```

parse_tree_children_no(N, "=", S, T, ["-", "-"], 2),
NN=N+1,
parse_tree_children_no(NN, "-", 1, S, [A, B], 2),
parse_tree_children_no(NN, "-", 2, S, [C, D], 2), !,
retract(parse_tree_children_no(N, "=", S, T, ["-", "-"], 2)),
retract_child(N, "=", S, T),
retract_children(NN, "-", 1, S),
retract_child(NN, "-", 1, S),
retract_children(NN, "-", 2, S),
retract_child(NN, "-", 2, S),
asserta(parse_tree_children_no(N, "=", S, T, ["-", "-"], 2)),
asserta(parse_tree_child_no(N, "=", S, T, "-", 1)),
asserta(parse_tree_child_no( N, "=", S, T, "-", 2)),
asserta(parse_tree_children_no(NN, "-", 1, S, [A, C], 2)),
asserta(parse_tree_child_no(NN, "-", 1, S, A, 1)),
asserta(parse_tree_child_no( NN, "-", 1, S, C, 2)),
asserta(parse_tree_children_no(NN, "-", 2, S, [B, D], 2)),
asserta(parse_tree_child_no(NN, "-", 2, S, B, 1)),
asserta(parse_tree_child_no(NN, "-", 2, S, D, 2)),
N1=NN+1,
checkA(N1, A, 1, 1, 1, 1),
checkA(N1, B, 2, 1, 1, 2),
checkA(N1, C, 1, 2, 2, 1),
checkA(N1, D, 2, 2, 2, 2).

```

op6:-

```

parse_tree_children_no(N, "=", S, T, ["-", "-"], 2),
NN=N+1,
parse_tree_children_no(NN, "-", 1, S, [A, B], 2),
parse_tree_children_no(NN, "-", 2, S, [C, D], 2), !,
retract(parse_tree_children_no(N, "=", S, T, ["-", "-"], 2)),
retract_child(N, "=", S, T),
retract_children(NN, "-", 1, S),
retract_child(NN, "-", 1, S),
retract_children(NN, "-", 2, S),
retract_child(NN, "-", 2, S),
asserta(parse_tree_children_no(N, "=", S, T, ["+", "+"], 2)),
asserta(parse_tree_child_no(N, "=", S, T, "+", 1)),
asserta(parse_tree_child_no( N, "=", S, T, "+", 2)),
asserta(parse_tree_children_no(NN, "+", 1, S, [A, D], 2)),
asserta(parse_tree_child_no(NN, "+", 1, S, A, 1)),
asserta(parse_tree_child_no( NN, "+", 1, S, D, 2)),
asserta(parse_tree_children_no(NN, "+", 2, S, [B, C], 2)),

```

```

asserta(parse_tree_child_no(NN, "+", 2, S, B, 1)),
asserta(parse_tree_child_no(NN, "+", 2, S, C, 2)),
N1=NN+1,
checkA(N1, A, 1, 1, 1, 1),
checkA(N1, B, 2, 1, 1, 2),
checkA(N1, C, 1, 2, 2, 2),
checkA(N1, D, 2, 2, 2, 1).

```

op6:-

```

parse_tree_children_no(N, "=", S, T, ["-", "+"], 2),
NN=N+1,
parse_tree_children_no(NN, "-", 1, S, [A, B], 2),
parse_tree_children_no(NN, "+", 2, S, [C, D], 2), !,
retract(parse_tree_children_no(N, "=", S, T, ["-", "+"], 2)),
retract_child(N, "=", S, T),
retract_children(NN, "-", 1, S),
retract_child(NN, "-", 1, S),
retract_children(NN, "+", 2, S),
retract_child(NN, "+", 2, S),
asserta(parse_tree_children_no(N, "=", S, T, ["-", "+"], 2)),
asserta(parse_tree_child_no(N, "=", S, T, "-", 1)),
asserta(parse_tree_child_no(N, "=", S, T, "+", 2)),
asserta(parse_tree_children_no(NN, "-", 1, S, [A, C], 2)),
asserta(parse_tree_child_no(NN, "-", 1, S, A, 1)),
asserta(parse_tree_child_no(NN, "-", 1, S, C, 2)),
asserta(parse_tree_children_no(NN, "+", 2, S, [B, D], 2)),
asserta(parse_tree_child_no(NN, "+", 2, S, B, 1)),
asserta(parse_tree_child_no(NN, "+", 2, S, D, 2)),
N1=NN+1,
checkA(N1, A, 1, 1, 1, 1),
checkA(N1, B, 2, 1, 1, 2),
checkA(N1, C, 1, 2, 2, 1),
checkA(N1, D, 2, 2, 2, 2).

```

op6:-

```

parse_tree_children_no(N, "=", S, T, ["-", "+"], 2),
NN=N+1,
parse_tree_children_no(NN, "-", 1, S, [A, B], 2),
parse_tree_children_no(NN, "+", 2, S, [C, D], 2), !,
retract(parse_tree_children_no(N, "=", S, T, ["-", "+"], 2)),
retract_child(N, "=", S, T),
retract_children(NN, "-", 1, S),
retract_child(NN, "-", 1, S),
retract_children(NN, "+", 2, S),
retract_child(NN, "+", 2, S),
asserta(parse_tree_children_no(N, "=", S, T, ["-", "+"], 2)),
asserta(parse_tree_child_no(N, "=", S, T, "-", 1)),
asserta(parse_tree_child_no(N, "=", S, T, "+", 2)),
asserta(parse_tree_children_no(NN, "-", 1, S, [A, D], 2)),
asserta(parse_tree_child_no(NN, "-", 1, S, A, 1)),
asserta(parse_tree_child_no(NN, "-", 1, S, D, 2)),
asserta(parse_tree_children_no(NN, "+", 2, S, [B, C], 2)),
asserta(parse_tree_child_no(NN, "+", 2, S, B, 1)),
asserta(parse_tree_child_no(NN, "+", 2, S, C, 2)),

```



```

N1=NN+1,
checkA(N1,A,1,1,1,1),
checkA(N1,B,2,1,1,2),
checkA(N1,C,1,2,2,2),
checkA(N1,D,2,2,2,1).

```

op6:-

```

parse_tree_children_no(N,"=",S,T,["+", "-"],2),
NN=N+1,
parse_tree_children_no(NN,"+",1,S,[A,B],2),
parse_tree_children_no(NN,"-",2,S,[C,D],2),!,
retract(parse_tree_children_no(N,"=",S,T,["+", "-"],2)),
retract_child(N,"=",S,T),
retract_children(NN,"+",1,S),
retract_child(NN,"+",1,S),
retract_children(NN,"-",2,S),
retract_child(NN,"-",2,S),
asserta(parse_tree_children_no(N,"=",S,T,["-", "+"],2)),
asserta(parse_tree_child_no(N,"=",S,T,"-",1)),
asserta(parse_tree_child_no(N,"=",S,T,"+",2)),
asserta(parse_tree_children_no(NN,"-",1,S,[C,A],2)),
asserta(parse_tree_child_no(NN,"-",1,S,C,1)),
asserta(parse_tree_child_no(NN,"-",1,S,A,2)),
asserta(parse_tree_children_no(NN,"+",2,S,[B,D],2)),
asserta(parse_tree_child_no(NN,"+",2,S,B,1)),
asserta(parse_tree_child_no(NN,"+",2,S,D,2)),
N1=NN+1,
checkA(N1,A,1,2,1,1),
checkA(N1,B,2,1,1,2),
checkA(N1,C,1,1,2,1),
checkA(N1,D,2,2,2,2).

```

op6:-

```

parse_tree_children_no(N,"=",S,T,["+", "-"],2),
NN=N+1,
parse_tree_children_no(NN,"+",1,S,[A,B],2),
parse_tree_children_no(NN,"-",2,S,[C,D],2),
retract(parse_tree_children_no(N,"=",S,T,["+", "-"],2)),
retract_child(N,"=",S,T),
retract_children(NN,"+",1,S),
retract_child(NN,"+",1,S),
retract_children(NN,"-",2,S),
retract_child(NN,"-",2,S),
asserta(parse_tree_children_no(N,"=",S,T,["+", "-"],2)),
asserta(parse_tree_child_no(N,"=",S,T,"+",1)),
asserta(parse_tree_child_no(N,"=",S,T,"-",2)),
asserta(parse_tree_children_no(NN,"+",1,S,[A,D],2)),
asserta(parse_tree_child_no(NN,"+",1,S,A,1)),
asserta(parse_tree_child_no(NN,"+",1,S,D,2)),
asserta(parse_tree_children_no(NN,"-",2,S,[C,B],2)),
asserta(parse_tree_child_no(NN,"-",2,S,C,1)),
asserta(parse_tree_child_no(NN,"-",2,S,B,2)),
N1=NN+1,
checkA(N1,A,1,1,1,1),
checkA(N1,B,2,2,1,2),
checkA(N1,C,1,2,2,2),

```

```
checkA(N1,D,2,1,2,1).
```

```
checkA(N,A,I,J,L,S):-  
  parse_leaf(N,A,I,L),!,  
  retract(parse_leaf(N,A,I,L)),  
  assertz(parse_leaf(N,A,J,S)).
```

```
checkA(N,A,I,J,L,S):-  
  parse_tree_children_no(N,A,I,L,ListA,M1),  
  retract(parse_tree_children_no(N,A,I,L,ListA,M1)),  
  M=M1+1,  
  include_child(N,A,I,L,J,S,ListA,1,M).
```

```
include_child(N,A,I,L1,J,S,L,M,M):-!,M1=M-1,  
  assertz(parse_tree_children_no(N,A,J,S,L,M1)).
```

```
include_child(N,A,I,L1,J,S,L,K,M):-  
  n_element(L,K,X),  
  retract(parse_tree_child_no(N,A,I,L1,X,K)),  
  assertz(parse_tree_child_no(N,A,J,S,X,K)),  
  NN=N+1,  
  checkA(NN,X,K,K,I,J),  
  KK=K+1,  
  include_child(N,A,I,L1,,S,L,KK,M).
```

```
/****** OP7 is the production for (ab/cd) is *****/  
/****** equivalent to (ab/c)/d *****/
```

```
op7:-
```

```
  parse_tree_children_no(N,"/",S,K,["*","*"],2),  
  NN=N+1,  
  parse_tree_children_no(NN,"*",1,S,[A,B],2),  
  parse_tree_children_no(NN,"*",2,S,[C,D],2),  
  N1=NN+1,  
  retract(parse_tree_children_no(N,"/",S,K,["*","*"],2)),  
  retract_child(N,"/",S,K),  
  retract_children(NN,"*",1,S),  
  retract_child(NN,"*",1,S),  
  retract_children(NN,"*",2,S),  
  retract_child(NN,"*",2,S),  
  retract(parse_leaf(N1,A,1,1)),  
  retract(parse_leaf(N1,B,2,1)),  
  retract(parse_leaf(N1,C,1,2)),  
  retract(parse_leaf(N1,D,2,2)),  
  N2=N1+1,  
  asserta(parse_leaf(N2,A,1,1)),  
  asserta(parse_leaf(N2,B,2,1)),  
  asserta(parse_leaf(N1,C,2,1)),  
  asserta(parse_leaf(NN,D,2,S)),  
  asserta(parse_tree_children_no(N,"/",S,K,["/",D],2)),  
  asserta(parse_tree_child_no(N,"/",S,K,["/",1]),
```

```

asserta(parse_tree_child_no(N, "/", S, K, D, 2)),
asserta(parse_tree_children_no(NN, "/", 1, 1, ["*", C], 2)),
asserta(parse_tree_child_no(NN, "/", 1, 1, "*", 1)),
asserta(parse_tree_child_no(NN, "/", 1, 1, C, 2)),
asserta(parse_tree_children_no(N1, "*", 1, 1, [A, B], 2)),
asserta(parse_tree_child_no(N1, "*", 1, 1, A, 1)),
asserta(parse_tree_child_no(N1, "*", 1, 1, B, 2)).

```

```

/***** OP8 is the production for (ab/cd) is *****/
/***** equivalent to (b/cd)*a *****/

```

op8:-

```

parse_tree_children_no(N, "/", S, K, ["*", "*"], 2),
NN=N+1,
parse_tree_children_no(NN, "*", 1, S, [A, B], 2),
parse_tree_children_no(NN, "*", 2, S, [C, D], 2),
N1=NN+1,
retract(parse_tree_children_no(N, "/", S, K, ["*", "*"], 2)),
retract_child(N, "/", S, K),
retract_children(NN, "*", 1, S),
retract_child(NN, "*", 1, S),
retract_children(NN, "*", 2, S),
retract_child(NN, "*", 2, S), NM=N-1,
parse_tree_child_no(NM, R, K, U, "/", I),
retract(parse_tree_child_no(NM, R, K, U, "/", I)),
asserta(parse_tree_child_no(NM, R, K, U, "*", I)),
parse_tree_children_no(NM, R, K, U, LL, MM),
retract(parse_tree_children_no(NM, R, K, U, LL, MM)),
replace_with(LL, "*", I, 1, [, KK),
asserta(parse_tree_children_no(NM, R, K, U, KK, MM)),
retract(parse_leaf(N1, A, 1, 1)),
retract(parse_leaf(N1, B, 2, 1)),
retract(parse_leaf(N1, C, 1, 2)),
retract(parse_leaf(N1, D, 2, 2)),
N2=N1+1,
asserta(parse_leaf(NN, A, 1, S)),
asserta(parse_leaf(N1, B, 1, 2)),
asserta(parse_leaf(N2, C, 1, 2)),
asserta(parse_leaf(N2, D, 2, 2)),
asserta(parse_tree_children_no(N, "*", S, K, [A, "/"], 2)),
asserta(parse_tree_child_no(N, "*", S, K, A, 1)),
asserta(parse_tree_child_no(N, "*", S, K, "/", 2)),
asserta(parse_tree_children_no(NN, "/", 2, S, [B, "*"], 2)),
asserta(parse_tree_child_no(NN, "/", 2, S, B, 1)),
asserta(parse_tree_child_no(NN, "/", 2, S, "*", 2)),
asserta(parse_tree_children_no(N1, "*", 2, 2, [C, D], 2)),
asserta(parse_tree_child_no(N1, "*", 2, 2, C, 1)),
asserta(parse_tree_child_no(N1, "*", 2, 2, D, 2)).

```

/***** This operator makes use of the operation, a=b gives b=a . *****/

op9:-

```
parse_tree_child_no(N, "=", S, K, A, 1),
parse_tree_child_no(N, "=", S, K, B, 2),
NN=N+1,
retract(parse_tree_children_no(N, "=", S, K, [A, B], 2)),
retract(parse_tree_child_no(N, "=", S, K, A, 1)),
retract(parse_tree_child_no(N, "=", S, K, B, 2)),
asserta(parse_tree_children_no(N, "=", S, K, [B, A], 2)),
asserta(parse_tree_child_no(N, "=", S, K, B, 1)),
asserta(parse_tree_child_no(N, "=", S, K, A, 2)),
checkA(NN, A, 1, 2, 1, 1),
checkA(NN, B, 2, 1, 1, 1).
```

/****** Representation of results at each node of the parse tree *****/
/*****

get_parse_result:-

```
parse_tree_children_no(1, R, S, K, L, M),
parse_tree_result(1, R, S, K).
```

parse_tree_result(N, R, S, K):-

```
parse_tree_children_no(N, R, S, K, List, M),
n_element(List, 1, X),
NN=N+1,
parse_leaf(NN, X, 1, S), !,
assertz(parse_result(NN, S, 1, X, X)),
concat(X, R, Y),
find_result(N, List, R, S, K, M, 1, Y).
```

parse_tree_result(N, R, S, K):-

```
parse_tree_children_no(N, R, S, K, List, M),
n_element(List, 1, X),
NN=N+1,
parse_tree_result(NN, X, 1, S),
parse_result(NN, S, 1, X, Z1),
concat(Z1, R, Y),
find_result(N, List, R, S, K, M, 1, Y).
```

find_result(N, L, R, S, K, M, I, Y):-

```
I1=I+1,
n_element(L, I1, X),
NN=N+1,
I1=M,
```

```

        parse_leaf(NN,X,I1,S),!,
        assertz(parse_result(NN,S,I1,X,X)),
        concat(Y,X,Z),
        store_parse_result(N,K,S,R,Z).

find_result(N,L,R,S,K,M,I,Y):-
    I1=I+1,
    n_element(L,I1,X),
    NN=N+1,
    I1=M,!,
    parse_tree_result(NN,X,I1,S),
    parse_result(NN,S,I1,X,Z1),
    concat(Y,Z1,Z),
    store_parse_result(N,K,S,R,Z).

find_result(N,L,R,S,K,M,I,Y):-
    I1=I+1,
    n_element(L,I1,X),
    NN=N+1,
    parse_leaf(NN,X,I1,S),!,
    concat(Y,X,Z),
    concat(Z,R,YY),
    find_result(N,L,R,S,K,M,I1,YY).

find_result(N,L,R,S,K,M,I,Y):-
    I1=I+1,
    n_element(L,I1,X),
    NN=N+1,
    parse_tree_result(NN,X,I1,S),
    parse_result(NN,S,I1,X,Z1),
    concat(Y,Z1,Z),
    concat(Z,R,YY),
    find_result(N,L,R,S,K,M,I1,YY).

store_parse_result(N,K,I,R,Z):-
    R="+",!,
    concat("(",Z,Y),
    concat(Y,")",P),
    asserta(parse_result(N,K,I,R,P)).

store_parse_result(N,K,I,R,Z):-
    R="-",!,
    concat("(",Z,Y),
    concat(Y,")",P),
    asserta(parse_result(N,K,I,R,P)).

store_parse_result(N,K,I,R,Z):-
    asserta(parse_result(N,K,I,R,Z)).

```

/*.....*/

```

/***** THIS IS AN IMPORTANT SECTION OF THE PROGRAM *****/
/*****MATCHING SECTION *****/
/*****

```

```

/**** After matching at the leaves, matching *****/
/**** at the internal nodes is done. *****/

```

```

match:-

```

```

    makewindow(3,7,7," MATCHING WINDOW ", 4,5,20,45),
    parse_tree_children_no(1,R,S,K,L,M1),
    M=M1+1,
    match_internal(1,R,S,K,1,M).

```

```

match_internal(N,R,S,K,M,M):-!.
match_internal(N,R,S,K,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    parse_leaf(NN,X,I,S),!,
    matching_leaf_list(NN,X,I,S,[]),
    II=I+1,
    match_internal(N,R,S,K,II,M).

```

```

match_internal(N,R,S,K,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    parse_tree_children_no(NN,X,I,S,L,MM),
    M1=MM+1,
    internal_match(NN,X,I,S,M1),
    II=I+1,
    match_internal(N,R,S,K,II,M).

```

```

internal_match(N,R,S,K,M):-
    parse_tree_child_no(N,R,S,K,X,1),
    NN=N+1,
    parse_leaf(NN,X,1,S),
    matching_leaf_list(NN,X,1,S,[]),
    matching_nodes(NN,X,1,S,SET1),!,
    repit(N,R,S,K,2,M,SET1).

```

```

internal_match(N,R,S,K,M):-
    parse_tree_child_no(N,R,S,K,X,1),
    NN=N+1,
    parse_tree_children_no(NN,X,1,S,L,M1),
    M2=M1+1,
    internal_match(NN,Y,1,S,M2),
    matching_nodes(NN,X,1,S,SET1),
    repit(N,R,S,K,2,M,SET1).

```

```

repit(N,R,S,K,M,M,SET1):-
    !,asserta(matching_nodes(N,R,S,K,SET1)).

```

```

repit(N,R,S,K,J,M,SET1):-
    parse_tree_child_no(N,R,S,K,Y,J),
    NN=N+1,
    parse_leaf(NN,Y,J,S),
    matching_leaf_list(NN,Y,J,S,[]),

```

```

    matching_nodes(NN, Y, J, S, SET2),
    R="*", !,
    intersect(SET1, SET2, SET),
    JJ=J+1,
    repit(N, R, S, K, JJ, M, SET) .
repit(N, R, S, K, J, M, SET1):-
    parse_tree_child_no(N, R, S, K, Y, J),
    NN=N+1,
    parse_leaf(NN, Y, J, S),
    matching_leaf_list(NN, Y, J, S, []),
    matching_nodes(NN, Y, J, S, SET2),
    R="/", !,
    intersect(SET1, SET2, SET),
    JJ=J+1,
    repit(N, R, S, K, JJ, M, SET) .
repit(N, R, S, K, J, M, SET1):-
    parse_tree_child_no(N, R, S, K, Y, J),
    NN=N+1,
    parse_leaf(NN, Y, J, S),
    matching_leaf_list(NN, Y, J, S, []),
    matching_nodes(NN, Y, J, S, SET2),
    R="+", !,
    union(SET1, SET2, SET),
    JJ=J+1,
    repit(N, R, S, K, JJ, M, SET) .
repit(N, R, S, K, J, M, SET1):-
    parse_tree_child_no(N, R, S, K, Y, J),
    NN=N+1,
    parse_leaf(NN, Y, J, S),
    matching_leaf_list(NN, Y, J, S, []),
    matching_nodes(NN, Y, J, S, SET2),
    R="-", !,
    union(SET1, SET2, SET),
    JJ=J+1,
    repit(N, R, S, K, JJ, M, SET) .

repit(N, R, S, K, J, M, SET1):-
    parse_tree_child_no(N, R, S, K, Y, J),
    NN=N+1,
    parse_tree_children_no(NN, Y, J, S, L, M1),
    M2=M1+1,
    internal_match(NN, Y, J, S, M2),
    matching_nodes(NN, Y, J, S, SET2),
    R="*", !,
    intersect(SET1, SET2, SET),
    JJ=J+1,
    repit(N, R, S, K, JJ, M, SET) .
repit(N, R, S, K, J, M, SET1):-
    parse_tree_child_no(N, R, S, K, Y, J),
    NN=N+1,
    parse_tree_children_no(NN, Y, J, S, L, M1),
    M2=M1+1,
    internal_match(NN, Y, J, S, M2),
    matching_nodes(NN, Y, J, S, SET2),
    R="/", !,
    intersect(SET1, SET2, SET),
    JJ=J+1,

```

```

        repit (N, R, S, K, JJ, M, SET) .
repit (N, R, S, K, J, M, SET1) :-
    parse_tree_child_no (N, R, S, K, Y, J) ,
    NN=N+1,
    parse_tree_children_no (NN, Y, J, S, L, M1) ,
    M2=M1+1,
    internal_match (NN, Y, J, S, M2) ,
    matching_nodes (NN, Y, J, S, SET2) ,
    R="+", !,
    union (SET1, SET2, SET) ,
    JJ=J+1,
    repit (N, R, S, K, JJ, M, SET) .
repit (N, R, S, K, J, M, SET1) :-
    parse_tree_child_no (N, R, S, K, Y, J) ,
    NN=N+1,
    parse_tree_children_no (NN, Y, J, S, L, M1) ,
    M2=M1+1,
    internal_match (NN, Y, J, S, M2) ,
    matching_nodes (NN, Y, J, S, SET2) ,
    R="-",
    union (SET1, SET2, SET) ,
    JJ=J+1,
    repit (N, R, S, K, JJ, M, SET) .

```

```

/*****THIS SECTION IS FOR MATCHING AT*****/
/*****LEAVES*****/
/*****

```

```

    matching_leaf_list (N, R, I, S, L) :-
        matched_leaf (R, M) , !,
        asserta (matching_nodes (N, R, I, S, M)) .

```

```

    matching_leaf_list (N, R, I, S, L) :-
        write ("Enter the process tree nodes that ") , nl,
        write ("match ' ", R, " ' : ") , nl,
        matching_nodes_list (N, R, I, S, L, M) ,
        asserta (matching_nodes (N, R, I, S, M)) ,
        asserta (matched_leaf (R, M)) .

```

```

    matching_nodes_list (N, R, I, S, List, Matchlist) :-
        readln (Matchnode) ,
        Matchnode <> " " , !,
        append_list (Matchnode, List, Newmatchlist) ,
        matching_nodes_list (N, R, I, S, Newmatchlist, Matchlist) .
    matching_nodes_list (N, R, I, S, List, List) .

```

```

/***** THIS PORTION OF THE PROGRAM IS FOR OUTPUT *****/
/***** AND FOR DISCARDING IRRELEVANT SUBPROCESSES. *****/

```



```

accumulation:-
    parse_tree_children_no(1,R,1,1,L,M),
    M1=M+1,
    accumulate(1,R,1,1,[],1,M1).

accumulate(N,R,S,K,L1,M,M):-!.
accumulate(N,R,S,K,L1,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    parse_leaf(NN,X,I,S),!,
    II=I+1,
    accumulate(N,R,S,K,L1,II,M).
accumulate(N,R,S,K,L1,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    matching_nodes(NN,X,I,S,L),
    length(L,1),!,
    parse_tree_children_no(NN,X,I,S,LIST,K1),
    KK=K1+1,
    accumulate(NN,X,I,S,L1,1,KK),
    II=I+1,
    accumulate(N,R,S,K,L1,II,M).

accumulate(N,R,S,K,L1,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    matching_nodes(NN,X,I,S,L),
    length(L,J),
    J>0,
    find_root(L,P),!,
    retract(matching_nodes(NN,X,I,S,L)),
    asserta(matching_nodes(NN,X,I,S,[P])),
    parse_tree_children_no(NN,X,I,S,LIST,K1),
    KK=K1+1,
    add(P,L1,L2),
    accumulate(NN,X,I,S,L2,1,KK),
    II=I+1,
    accumulate(N,R,S,K,L1,II,M).

accumulate(N,R,S,K,L1,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    matching_nodes(NN,X,I,S,LK),
    deduct(LK,L1,L),
    length(L,1),!,
    retract(matching_nodes(NN,X,I,S,LK)),
    asserta(matching_nodes(NN,X,I,S,L)),
    parse_tree_children_no(NN,X,I,S,LIST,K1),
    KK=K1+1,
    accumulate(NN,X,I,S,L1,1,KK),
    II=I+1,
    accumulate(N,R,S,K,L1,II,M).

accumulate(N,R,S,K,L1,I,M):-
    operator_search,

```

```

retract_result,
retract_match,
get_parse_result,
match,
accumulation.

```

```

/***** THIS PORTION OF THE PROGRAM IS FOR OUTPUT *****/
/***** AND FOR DISCARDING IRRELEVANT SUBPROCESSES.*****/

```

```

assertion:-

```

```

    makewindow(4,7,7,"  OUTPUT SCREEN      ",4,5,18,50),
    parse_tree_children_no(1,R,1,1,L,M),
    M1=M+1,
    speak(1,R,1,1,1,M1).
speak(N,R,S,K,M,M):-!.
speak(N,R,S,K,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    parse_leaf(NN,X,I;S),!,
    II=I+1,
    speak(N,R,S,K,II,M).
speak(N,R,S,K,I,M):-
    parse_tree_child_no(N,R,S,K,X,I),
    NN=N+1,
    matching_nodes(NN,X,I,S,L),
    parse_result(NN,S,I,X,Y),
    write("The equation at the node",L,"is:",Y),nl,
    write("Press Enter: "),
    readln(A),
    A="",
    nl,
    parse_tree_children_no(NN,X,I,S,LIST,K1),
    KK=K1+1,
    speak(NN,X,I,S,1,KK),
    II=I+1,
    speak(N,R,S,K,II,M).
speak(N,R,S,K,I,M).

```

```

/***** This section analyses the process tree for match*****/
/***** and uses the helping predicates*****/

```

```

whether_same_process_branch:-
    process_tree_children_no(1,R,1,L,M),
    M1=M+1,
    same_branch(L,1,M1).

```

```

same_branch(L,M,M):-!.
same_branch(L,I,M):-

```

```

n_element(L, I, X),
same_branch_check(X),
II=I+1,
same_branch(L, II, M).

same_branch_check(X):-
    process_leaf(N, X, I), !,
    asserta(same_process_branch(X, [X])).

same_branch_check(X):-
    process_tree_children_no(N, X, I, L, M),
    M1=M+1,
    next_check(X, L, [], 1, M1).

next_check(X, L, L1, M1, M1):-!,
    append([X], L1, LL),
    asserta(same_process_branch(X, LL)).
next_check(X, L, L1, I, M1):-
    n_element(L, I, Y),
    same_branch_check(Y),
    same_process_branch(Y, YL),
    append(L1, YL, LL),
    II=I+1,
    next_check(X, L, LL, II, M1).

equal_list([], []):-!.
equal_list(L, LL):-
    length(L, K),
    length(LL, K),
    n_element(L, 1, X),
    member(X, LL),
    pull(X, L, L1),
    pull(X, LL, L2),
    equal_list(L1, L2).

add(P, L, LL):-
    member(P, L), !,
    LL=L.
add(P, L, LL):-
    append([P], L, LL).

deduct(LL, [], LL):-!.
deduct(LL, [H|T], L1):-
    pull(H, LL, L2),
    deduct(L2, T, L1).

find_root(L, P):-
    same_process_branch(P, LL),
    equal_list(L, LL).

```

```

length([],0):-!.
length(_|R|,L):-
    length(R,L1),
    L=L1+1.

retract_children(N,A,I,K):-
    retract(parse_tree_children_no(N,A,I,K,_,_)),fail.
retract_children(N,A,I,K).

retract_child(N,A,I,K):-
    retract(parse_tree_child_no(N,A,I,K,_,_)),fail.
retract_child(N,A,I,K).

retract_result:-
    retract(parse_result(?,?,?,?)),fail.
retract_result.

retract_match:-
    retract(matching_nodes(?,?,?,?)),fail.
retract_match.

union([],SET2,SET2):-
    !.
union([H|T],SET2,LIST):-
    member(H,SET2),!,
    union(T,SET2,LIST).
union([H|T],SET2,LIST):-
    append(SET2,[H],NEW),
    union(T,NEW,LIST).

intersect([],REM,[]):-
    !.
intersect(REM,[],[]):-
    !.
intersect([H|T1],SET2,LIST):-
    member(H,SET2),
    pull(H,SET2,REM),
    intersect(T1,SET2,NEW),!,
    append([H],NEW,LIST).

```

```
intersect ([H|T], SET2, LIST):-
    intersect (T, SET2, LIST).
```

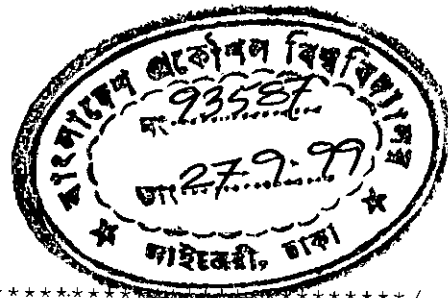
```
append ([], LISTB, LISTB).
append ([X|LIST1], LIST2, [X|LIST3]):-
    append (LIST1, LIST2, LIST3).
```

```
member (X, [X|_]) :-!.
member (X, [_|TAIL]):-
    member (X, TAIL).
```

```
pull (X, [X|List], List):-!.
pull (X, [OTHER|REM], [OTHER|REM1]):-
    pull (X, REM, REM1).
```

```
check (A):-
    A="Y",!.
check (A):-
    A="y".
```

```
response (Reply):-
    readln (Reply),
    write (Reply), nl.
```



```
/*
*****
*****THEEND*****
*****
*/
```