INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University
Microfilms
International

300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 BEDFORD ROW, LONDON WC1R 4EJ, ENGLAND

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark ✓.

1. Glossy photographs _____
2. Colored illustrations _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Print shows through as there is text on both sides of page _____
6. Indistinct, broken or small print on several pages ✓ throughout ✓
7. Tightly bound copy with print lost in spine _____
8. Computer printout pages with indistinct print _____
9. Page(s) _____ lacking when material received, and not available from school or author _____
10. Page(s) _____ seem to be missing in numbering only as text follows _____
11. Poor carbon copy _____
12. Not original copy, several pages with blurred type _____
13. Appendix pages are poor copy _____
14. Original copy with light type _____
15. Curling and wrinkled pages _____
16. Other _____

On the Efficient Implementation of Production Systems (Thesis Summary)

Charles L. Forgy

30 January 1979

Carnegie-Mellon University
Department of Computer Science

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Carnegie-Mellon University

MELLON INSTITUTE OF SCIENCE

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE ON THE EFFICIENT IMPLEMENTATION OF PRODUCTION SYSTEMS

PRESENTED BY Charles Lanny Forgy

ACCEPTED BY THE DEPARTMENT OF Computer Science

Allen Newell — MAJOR PROFESSOR — 7 Feb 79 — DATE

William Wulf — DEPARTMENT HEAD — 7 Feb 1979 — DATE

APPROVED BY

DEAN — 4-3-79 — DATE

Abstract

It is not uncommon for an Artificial Intelligence program to spend most of its time evaluating patterns in order to locate either subprograms or entries in a data base. This is particularly true of production systems since, unlike other programs, they have no alternatives to pattern evaluation. Other programs may call some functions by name or access some data by retrieving the bindings of variables, but a production system uses pattern evaluation to select every procedure it executes and to locate every piece of data operated upon by the procedures. In this thesis, methods are described which can greatly reduce the amount of time that Artificial Intelligence programs like production systems spend in pattern evaluation.

The thesis is concerned both with the algorithms used by a production system interpreter and with the hardware on which the algorithms are executed. The thesis contains a detailed description of a method for evaluating a set of patterns which (1) notes the similarities in the patterns so that it can avoid performing the same test more than once; (2) takes advantage of the fact that both the set of patterns and the set of objects change slowly by saving information from one evaluation to the next; and (3) allows a high degree of parallel activity during the evaluation. This method involves the use of a compiler which translates the patterns into a program for a virtual pattern-matching machine. It is shown in the thesis that, although the instructions for this machine appear quite different from the instructions for a conventional processor, they can be interpreted efficiently on a conventional microprogrammed computer. If a microprogrammed computer were augmented with some inexpensive hardware described in the thesis, it would be able to interpret the virtual machine instructions as fast as it interprets conventional instructions. Without the special hardware, the computer would interpret the virtual machine instructions about three times more slowly.

This thesis contains an analytical study of the pattern-matching algorithm and an empirical study of an interpreter which uses a Lisp implementation of the algorithm. These studies showed that the time required to evaluate the patterns would vary with the logarithm of the number of patterns. An interpreter running on a microprogrammed processor should be about two orders of magnitude faster than current interpreters; an interpreter running on the proposed special hardware should be about two and one-half orders of magnitude faster than current interpreters. The studies showed also that the space required to store the compiled patterns would be a linear function of the number of patterns. The compiled patterns would be smaller than the uncompiled patterns by a factor of perhaps two.

Acknowledgments

I wish to thank

Allen Newell, my thesis advisor, for supervising my work during the past several years.

Bob Sproull for valuable discussions concerning the research and for serving on the thesis committee after the research was completed.

Sam Fuller and Raj Reddy, the other two members of my thesis committee, for thoughtful comments on an earlier version of this document.

John Laird, Jill Larkin, John McDermott, and Michael Rychener for helping me to make measurements of their programs.

Diana Connan Forgy for her patience and for helping to produce this document.

Table of Contents

1. Introduction

A production system is a programming language with an unfortunate characteristic: larger production system programs execute more slowly than small ones. The extra instructions in the larger program do not have to execute to slow down the system; their mere presence is sufficient. One can take a working program, add a collection of instructions that will not take part in the task it performs, and thereby slow down the program. Yet production systems have another property that makes them particularly attractive for constructing large programs: they do not require the programmer to specify in minute detail exactly how the various parts of the program will interact. There is thus reason to try to reduce the dependency of execution speed on program size. This thesis reports the results of one study which tried to reduce the dependency by developing better algorithms for the production system interpreter and by developing hardware on which the algorithms could run efficiently.

1.1 The Problem

A production system interpreter is a computer comprising a processor plus two disjoint memories called production memory (or PM) and working memory (WM). Production memory holds the program executed by the processor, and working memory holds the data operated on by the program. The objects contained in working memory are called data elements. A data element is a symbol structure -- that is, a word in a formal language like the language of lists (see section 1.2.1) or the language of name-attribute-value triples. If a production system was interested in Greek tragedy, working memory might contain these data elements:

```
Agenor is-father-of Cadmus,
Cadmus is-father-of Polydorus,
Polydorus is-father-of Labdacus,
Labdacus is-father-of Laius,
```

The objects contained in production memory are condition-action pairs called productions.

```
If there is a man whose father is Laius,
Then assert that the man killed Laius,
```

The conditions are conditions on the contents of working memory, and the actions are

primarily commands to change the contents of working memory. The two most common kinds of change are adding one new data element to working memory and deleting one existing element.

Unlike a conventional computer program, a production system incorporates no concept of sequential flow of control through the program. The flow of control in a production system is determined by the order in which the condition parts of the productions become true. The interpreter repeatedly executes the following steps.

1. Determine which productions have true condition parts.
2. If there are no productions with true condition parts, halt the system; otherwise, select one production from those that do.
3. Perform the actions specified by the chosen production.
4. Go to step 1.

This sequence is called the recognize-act cycle. Step (1) of the cycle is called the match. Step (2) is called conflict resolution, and step (3) is called act.

The problem considered in this thesis is how to perform the match efficiently. Production systems have historically operated from one to two orders of magnitude slower than conventional programs, due in large part to the difficulty of performing the match. Moreover, unless production system interpreters are improved, this performance gap will widen. Since production systems are being applied to ever more complex tasks, the size of the average production system is increasing. As is perhaps already obvious, the cost of performing the match depends on both the number of productions in production memory and the number of data elements in working memory. This thesis attempted to develop methods for the match which would allow even very large production systems to compete in efficiency with programs written in other languages.

1.2 OPS2

The remainder of this chapter is devoted to background material, including a more detailed description of the problem and a survey of previous work in the area. This section begins the background material by describing OPS2, the production system language that will be used in all the examples in this thesis. This section describes only those features of the language which are needed to understand the examples; a complete description is available in Forgy and McDermott [22].

1.2.1 Lisp

OPS2 is implemented in the language Lisp, and OPS2 production systems make use of Lisp data types. Thus while one need not know how to program in Lisp to understand an OPS2 production system, one does need to be familiar with Lisp data types. This section provides a brief introduction to the two most important Lisp data types, lists and atoms.[1]

There are two kinds of atoms, numeric atoms and literal atoms. A literal atom is a string of one or more letters, digits, and special characters like "%" or "#" (the special characters that are allowed differ from one Lisp system to the next). The following are legal Lisp atoms.

```
A
=
#X
MONKEY
Monkey
-->
C12H22011
```

A numeric atom is an integer or a floating point number.

```
0
-1535
2.7183
```

A list is a sequence of zero or more atoms or other lists enclosed in parentheses.[2] Adjacent atoms are separated by spaces, tabs, line feeds, or the like.

```
(A)
()
(-1535)
(MONKEY HOLDS BANANAS)
((1) (2) (3))
( (Want =Z) =Z --> (<DELETE> (Want =Z)) )
```

---

[1] For descriptions of the other Lisp data types plus an explanation of how to program in Lisp, see Weissman [60] or Siklossy [53].

[2] This definition does not capture the entire meaning of the term "list"; much more complex structures can be built. The simple definition used here is sufficient to understand OPS2.

The concept of equality of objects is very important in OPS2; hence it is necessary to consider what it means for two Lisp objects to be equal.[1] Equality of literal atoms is simple to define; two literal atoms are equal only if they comprise identical strings of characters. Thus **AB** is equal to **AB**, but not to **BA**, **AAB**, or **ABB**. The atom **MONKEY** is not equal to **Monkey**. Two numeric atoms are equal if their algebraic difference is zero. When an integer is compared to a floating point number, the integer is converted to floating point. Thus **3** is equal to **3.0** but not to **3.01**. Two lists are equal if their corresponding parts are equal; it is necessary that they have the same number of parts and that the parts be disposed identically. The list **(A B 1)** is equal to **(A B 1)** and **(A B 1.0)**, but not to **(A 1 B)**, **(A B 1 1)**, or **((A B 1))**. *Atoms are not broken into their constituent characters, so* **(A B 1)** *is not equal to* **(AB 1)** or **(AB1)**.

1.2.2 Match

A production is a list containing a right pointing arrow (the atom **-->**). The condition part of a production is the part before the arrow; it is called the LHS (left hand side) of the production. Both working memory and the LHSs of the productions contain lists. The lists in the productions, which are called condition elements, are patterns. The match tries to find instances of the class defined by the pattern among the lists in working memory -- a process called instantiating the pattern. A production is ready to be executed when all its condition elements are instantiated. (An exception to this is explained below in the paragraph about the special symbol "-".) The ordered pair of a production name and the collection of data elements that instantiate the production's condition elements is called an instantiation. The responsibility of the match is to find the set of all legal instantiations (the conflict set).

A condition element can contain variables and functions to be executed during the match. A condition element can be instantiated by a working memory element if the two can be made equal by substituting appropriate quantities for the variables and functions. This section describes the variables and functions that can appear in condition elements.

The simplest condition element is a constant or a list of constants:

```
Stop
(Monkey On Ladder)
(Want (Monkey On Ladder))
```

The match is allowed to make no substitutions for a constant; a data element can instantiate a

[1] For those familiar with Lisp, the definition of equality used here is that used by the predicate EQUAL.

constant condition element only if the two elements are equal.

Somewhat less simple are the condition elements containing variables. A variable in an OPS2 production system is a literal atom beginning with =, #, <, or >.

```
=W
=P
#P
```

Variables beginning with = are the most important; these are the ordinary variables found in most pattern matching systems. The match is allowed to substitute any data subelement for such a variable. The variable is said to be bound to the data element that substitutes for it. If a variable occurs more than once in a LHS, all occurrences must be bound to the same subelement. A typical use of variables is seen in production MB10.[1]

```
MB10    ( (Want =Z) =Z --> (<DELETE> (Want =Z)) )
```

This production has two condition elements, (Want =Z) and =Z. If working memory held only the following three elements

```
(Want (Monkey Holds Bananas))
(Want (Monkey Near (8 2)))
(Monkey Near (8 2))
```

then MB10 would have one instantiation:

```
<MB10, (Want (Monkey Near (8 2))) (Monkey Near (8 2))>.
```

(Recall that an instantiation is defined to be an ordered pair of a production name and the data elements that instantiate the production's condition elements.) The list (Monkey Near (8 2)) is bound to both occurrences of =Z.

As variables beginning with = indicate that two elements must be equal, variables beginning with # indicate that two elements must be unequal. A variable beginning with # may occur in a LHS only if the LHS contains another variable beginning with =, but otherwise spelled identically. The variable #P can occur in MB8, for example, only because =P occurs there also.

---

[1] The productions and most of the condition elements shown in this section are taken from the MKYBAN production system, which will be described in section 1.2.6.

```
MB8     ( (Want (=0 Near =P)) (Light =0) (=0 Near #P)
          -->
          (Want (Monkey Holds =0))  )
```

The variables beginning with # can be substituted for by any data subelement, but the subelement must not be equal to the subelement bound to the = variable.

Variables beginning with < and > are are used much like those beginning with #. These also are allowed only if elsewhere in the LHS there is a variable beginning with = but otherwise spelled identically. The variables beginning with < can be substituted for by numeric atoms smaller than those bound to the = variable; the variables beginning with > can be substituted for by numeric atoms greater than those bound to the = variable.

One variable, = by itself, has been defined to be a "don't care". The variable = can be substituted for by any data subelement, and there is no requirement that multiple occurrences must be substituted for by the same subelement. Thus the condition element

```
(= = =)
```

would match any data element with three subelements, including both of the following.

```
(1 1 1)
(A B C)
```

Ordinarily a condition element can be instantiated only by a data element having the same number of subelements. Two special symbols are provided in OPS2 to relax this restriction. The symbol ! allows substituting an arbitrarily long tail of a list for a single condition subelement. The condition element

```
(A ! =)
```

could, for example, be instantiated by any of the following.

```
(A 1)
(A 1 2)
(A (A))
```

Since the tail can have length zero, the condition element could also be instantiated by a list containing only one element.

```
(A)
```

The other special character is &. It is used to group together two condition elements so that they can be substituted for by a single data element. This character is used in the last condition element of MB12.

```
MB12    ( (Want (Monkey Near =P)) (Monkey On Floor)
          (Monkey Near =C & #P)
          -->
          (<WRITE> "The monkey walks from" =C "to" =P)
          (<DELETE> (Want (Monkey Near =P)) )
          (<DELETE> (Monkey Near =C) )
          (Monkey Near =P) )
```

Since the & groups together =C and #P, this condition element can be instantiated only by data elements with three subelements.

OPS2 provides the user a limited facility for defining his own tests. The user writes a Lisp program to test data subelements and declares to the interpreter that it is for use in the match. After that, when the interpreter tries to instantiate a condition element containing the function name, it will apply the function to the data subelements that substitute for the function name. Only those substitutions for which the function returns TRUE will be allowed. The following condition element, for example, contains an occurrence of the function **<NOTANY>**.

```
(Monkey On =O & (<NOTANY> Floor))
```

As this shows, arguments to the function may be included in the LHS. The **<NOTANY>** function disallows constants that are equal to its argument. In this case its argument is **Floor**, so the condition element can be instantiated by

```
(Monkey On Couch)
```

or

```
(Monkey On Ladder)
```

but not

```
(Monkey On Floor).
```

The special symbol - allows the user to specify that a condition element must not be instantiated. A LHS is satisfied when all of the condition elements not preceded by - are

instantiated and none of the condition elements preceded by - are instantiated. The bindings to variables are considered in determining whether the condition elements are instantiated. Since the second condition element of MB19 is negated,

```
MB19    ( (Want (EmptyHanded =X)) - (=X Holds =)
          -->
          (<DELETE> (Want (EmptyHanded =X)) ) )
```

if working memory held only the following three elements

```
(Want (EmptyHanded Monkey1))
(Want (EmptyHanded Monkey2))
(Monkey1 Holds Ladder)
```

then MB19 would have one instantiation:

```
<MB19, (Want (EmptyHanded Monkey2))>.
```

1.2.3 Conflict Resolution

Conflict resolution performs two functions. First it determines whether execution of the production system should halt, and then if not, it chooses the one instantiation to be executed in act. It performs these functions by applying in order up to seven rules called conflict resolution rules. The rules are

1. Halt the system if the conflict set is empty.

2. Exclude from consideration all instantiations that have previously executed. If none are left after this step, halt the system.

3. Order the instantiations based on the recency of the data elements they contain and then exclude from consideration all instantiations except the ones dominating under this order. In performing this comparison, the interpreter considers all the data elements of the instantiations. To order two instantiations it first compares their most recent elements. If one is more recent than the other, the instantiation containing the more recent element dominates. If both elements are equally recent, the interpreter compares the second most recent elements of both instantiations. If they are equally recent, it compares the third most recent elements, and so on. If the data elements of one instantiation are exhausted, the instantiation not exhausted dominates. Only if the two instantiations are exhausted simultaneously does this rule consider them equal.

4. If more than one instantiation remains after applying the third rule, eliminate all the instantiations whose productions have fewer condition elements than others in the remaining set.

5. If more than one instantiation remains after applying the fourth rule, eliminate all the instantiations whose productions have fewer constant atoms in their LHSs than others in the remaining set.

6. If more than one instantiation remains after applying the fifth rule, eliminate all the instantiations except those of the most recently created production in the remaining set.

7. If more than one instantiation remains after applying the sixth rule, make an arbitrary selection of the instantiation to execute.

For a justification of this rule set, see McDermott and Forgy [34].

1.2.4 Act

The part of an OPS2 production following the arrow is called the RHS (right hand side). An RHS contains zero or more lists called actions. In the act part of the recognize-act cycle, one production is executed by performing all its actions.

The simplest action is a list of constant atoms.

```
(Want (EmptyHanded Monkey))
```

Actions of this kind are executed by copying the list and then adding the resulting list to working memory.

The next step in complexity is an action containing variables that were bound during the match.

```
(Want (Ladder Near =P))
(=Z Near =P)
```

Actions of this kind are executed by copying the list, replacing the variables by the values to which they were bound, and then adding the resulting list to working memory. Executing the first action above, for example, might result in adding to working memory the list

```
(Want (Ladder Near (8 2))).
```

The final step in complexity are the actions containing RHS functions. An RHS function is a Lisp function that is called when the action containing it is executed. Some of the RHS functions are like variables in that they affect the copying of the list. One very common function of this class is the action <READ>, which accepts input from the user and inserts it

into the list in the place of the function. For example, if the action

```
(INPUT (<READ>))
```

were executed, and if the user typed in

```
HALT
```

working memory would have added to it the data element

```
(INPUT HALT).
```

Another kind of RHS action is executed not to affect the copying of a list, but to realize some side effect. Perhaps the most important function in this class is <DELETE>. This function accepts one or more arguments which it copies in the usual fashion, replacing variables and executing RHS functions. After the lists are copied, it searches working memory to determine if identical elements are contained there; if it finds any, it removes them from the memory.

```
(<DELETE> (Want =Z))
(<DELETE> (Want (Monkey Near =P)))
```

The action <WRITE> is also executed only for its side effects. This function copies its arguments like <DELETE> and then prints them on the user's terminal.

1.2.5 Self Modification

Two RHS functions are provided to allow production systems to modify themselves while they are running. The action <BUILD> evaluates its argument in the usual way and then adds the result to production memory. The argument must, of course, evaluate to a legal production. The following production, for example, would pick up productions that had been assembled in working memory and add them to production memory.

```
PBUILD  ( (Production =P) --> (<BUILD>  =P) )
```

The <BUILD> action accepts an optional name for the production. If no name is provided, <BUILD> creates one. In either case, <BUILD> returns the name of the new production. The action <EXCISE> is the inverse of <BUILD>. Its argument should evaluate to the name of a production. That production is removed from production memory.

1.2.6 The MKYBAN Production System

This section contains the source listing of a production system named MKYBAN, which solves the venerable monkey and bananas problem. This production system gives a monkey the abilities to move around a room, to climb onto and off of objects, to grasp and release objects, and to carry light objects as it moves. Goals can be set for the monkey, and the production system will try to satisfy the goals using the abilities of the monkey. Three problems that the system can solve are listed after the production system. The three problems are all variants of the task of getting bunch of bananas that are too high for the monkey to reach.

Before showing the production system, the coding conventions used in the system should be described. Working memory in the system contains only two kinds of elements. Most of the elements are assertions about the current status of the system.

```
(Monkey On Couch)
(Monkey Holds Ladder)
(Bananas Near (8 2))
```

The rest of the data elements are descriptions of states that the system is trying to achieve.

```
(Want (Monkey Holds Bananas))
(Want (Monkey On Ladder))
```

The productions' LHSs are all similar. Each LHS begins with a condition element to match one of these "wants". The rest of the condition elements pick up the data needed by the RHS actions.

The following is a listing of MKYBAN as it would be read into the OPS2 interpreter. Lines beginning with "~" are comments; everything else is input to the interpreter.

```
[SYSTEM

~       How to get objects that are too high to reach

MB1     ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
        -->
        (Want (Ladder Near =P)) )

MB2     ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
        (Ladder Near =P)
        -->
        (Want (Monkey On Ladder)) )
```

```
MB3    ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
       (Ladder Near =P) (Monkey On Ladder)
       -->
       (Want (EmptyHanded Monkey)) )

MB4    ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
       (Ladder Near =P) (Monkey On Ladder) - (Holds Monkey =)
       -->
       (<WRITE> "The monkey grabs the" =W)
       (Monkey Holds =W) (<DELETE> (Want (Monkey Holds =W))) )

~      How to get objects that are low enough to reach

MB5    ( (Want (Monkey Holds =W)) - (High =W) (=W Near =P)
       -->
       (Want (Monkey Near =P)) )

MB6    ( (Want (Monkey Holds =W)) - (High =W) (=W Near =P)
       (Monkey Near =P)
       -->
       (Want (EmptyHanded Monkey)) )

MB7    ( (Want (Monkey Holds =W)) - (High =W) (=W Near =P)
       (Monkey Near =P) - (Holds Monkey =)
       -->
       (<WRITE> "The monkey grabs the" =W)
       (Monkey Holds =W) (<DELETE> (Want (Monkey Holds =W))) )

~      How to move light objects from place to place

MB8    ( (Want (=O Near =P)) (Light =O) (=O Near =P)
       -->
       (Want (Monkey Holds =O)) )

MB9    ( (Want (=O Near =P)) (Light =O) (=O Near =P)
       (Monkey Holds =O)
       -->
       (Want (Monkey Near =P)) )

MB10   ( (Want =Z) =Z --> (<DELETE> (Want =Z)) )

~      How the monkey moves from place to place

MB11   ( (Want (Monkey Near =P))
       -->
       (Want (Monkey On Floor)) )

MB12   ( (Want (Monkey Near =P)) (Monkey On Floor)
       (Monkey Near =C & =P)
       -->
       (<WRITE> "The monkey walks from" =C "to" =P)
       (<DELETE> (Want (Monkey Near =P)) )
       (<DELETE> (Monkey Near =C) )
       (Monkey Near =P) )

MB13   ( (Want (Monkey Near =P)) (Monkey On Floor)
       (Monkey Near =C & =P) (Monkey Holds =Z)
       -->
       (<WRITE> "The monkey walks from" =C "to" =P)
       (<DELETE> (Want (Monkey Near =P)) )
       (<DELETE> (Monkey Near =C) (=Z NEAR =C) )
       (Monkey Near =P) (=Z Near =P) )
```

```
~       How to climb on and off objects

MB14    ( (Want (Monkey On Floor))
        (Monkey On =O & (<NOTANY> Floor))
        -->
        (<WRITE> "The monkey jumps off of the" =O)
        (<DELETE> (Want (Monkey On Floor)) )
        (<DELETE> (Monkey On =O))
        (Monkey On Floor) )

MB15    ( (Want (Monkey On =O)) (=O Near =X)
        -->
        (Want (Monkey Near =X)) )

MB16    ( (Want (Monkey On =O)) (=O Near =X) (Monkey Near =X)
        -->
        (Want (EmptyHanded Monkey)) )

MB17    ( (Want (Monkey On =O)) (=O Near =X) (Monkey Near =X)
        - (Monkey Holds =) (Monkey On =Z)
        -->
        (<WRITE> "The monkey climbs onto the" =O)
        (<DELETE> (Want (Monkey On =O)) )
        (<DELETE> (Monkey On =Z) )
        (Monkey On =O) )

~       How the monkey empties his hands

MB18    ( (Want (EmptyHanded Monkey)) (Monkey Holds =X)
        -->
        (<WRITE> "The monkey drops the" =X)
        (<DELETE> (Want (EmptyHanded Monkey)) )
        (<DELETE> (Monkey Holds =X) ) )

~       To recognize when the monkey is already EmptyHanded

MB19    ( (Want (EmptyHanded =X)) - (=X Holds =)
        -->
        (<DELETE> (Want (EmptyHanded =X)) ) )
]

~ T1, T2, and T3, defined below, are sample problem statements

[DV T1 ((Want (Monkey Holds Bananas))
       (Monkey Near (5 7))
       (Monkey On Couch)
       (Couch Near (5 7))
       (Bananas Near (8 2))
       (High Bananas)
       (Light Ladder)
       (Ladder Near (2 2))
]

[DV T2 ((Want (Monkey Holds Bananas))
       (Monkey Near (2 2))
       (Monkey On Floor)
       (Couch Near (5 7))
       (Bananas Near (8 2))
       (High Bananas)
       (Light Ladder)
       (Ladder Near (2 2))
]
```

```
[DV T3 ((Want (Monkey Holds Bananas))
       (Monkey Near (5 7))
       (Monkey On Couch)
       (Couch Near (5 7))
       (Bananas Near (8 2))
       (High Bananas)
       (Light Ladder)
       (Ladder Near (8 2))
]
```

1.2.7 Execution of MKYBAN

The following is a trace of the MKYBAN production system solving one of the problems defined in the source file. Commentary on the trace is in italics.

```
>
```

The interpreter prompts with >. Everything shown here is output from the computer except the underlined command below to start processing problem T3.

```
>(START T3)
```

The elements describing problem T3 are forced into working memory, causing MKYBAN to begin processing. Each time a production fires, the interpreter prints the data image of the production (i.e., the data to which the LHS and RHS are instantiated).

```
MB2
  (Want (Monkey Holds Bananas))(High Bananas)(Bananas Near (8 2))
  (Ladder Near (8 2))
 -->
    (Want (Monkey On Ladder))
```

It is possible to infer the general structure of a production from the trace information. The production that just fired, for example, is

```
MB2     ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
          (Ladder Near =P)
          -->
          (Want (Monkey On Ladder)) )
```

```
MB15
  (Want (Monkey On Ladder))(Ladder Near (8 2))
 -->
    (Want (Monkey Near (8 2)))

MB11
  (Want (Monkey Near (8 2)))
 -->
    (Want (Monkey On Floor))
```

The trace of MB14 shows two new kinds of information. Before the name of the production is what was printed by a <WRITE> action. After the element that was added to working memory are the two elements that were deleted.

```
The monkey jumps off of the Couch
MB14
  (Want (Monkey On Floor))(Monkey On Couch)
 -->
    (Monkey On Floor)
 Deleted:
    (Want (Monkey On Floor))(Monkey On Couch)

The monkey walks from (5 7) to (8 2)
MB12
  (Want (Monkey Near (8 2)))(Monkey On Floor)(Monkey Near (5 7))
 -->
    (Monkey Near (8 2))
 Deleted:
    (Want (Monkey Near (8 2)))(Monkey Near (5 7))

The monkey climbs onto the Ladder
MB17
  (Want (Monkey On Ladder))(Ladder Near (8 2))(Monkey Near (8 2))
  (Monkey On Floor)
 -->
    (Monkey On Ladder)
 Deleted:
    (Want (Monkey On Ladder))(Monkey On Floor)
```

```
The monkey grabs the Bananas
MB4
  (Want (Monkey Holds Bananas))(High Bananas)(Bananas Near (8 2))
  (Ladder Near (8 2))(Monkey On Ladder)
 -->
    (Monkey Holds Bananas)
 Deleted:
    (Want (Monkey Holds Bananas))
```

With the execution of MB4, the solution of the problem is complete. Since no other productions have satisfied LHSs, the system halts.

```
END -- NO PRODUCTION TRUE
```

1.3 The Scope of the Problem

While the discussion in this thesis must be restricted to a single production system language if it is to be kept short and focused, a potential problem arises from so doing. The reader might receive the impression that this thesis is concerned just with the design of an interpreter for OPS2. Quite the contrary is true, however. The goal of the research described here has been to develop techniques that can be applied to all the production systems in the class described in section 1.1. And there is reason to believe that the techniques can easily be adapted to languages outside this class. Chapter 2, which introduces the pattern-matching algorithms, ends with an argument that the algorithms would be widely applicable. This section surveys the languages that use pattern matching.

1.3.1 Other Languages Using Pattern Matching

The survey of languages in this section is rather brief. For other surveys and general discussions of production systems, see Newell [37, 39], Davis and King [14], Waterman and Hayes-Roth [59], and Hayes-Roth, Waterman, and Lenat [25]. For a survey of the other Artificial Intelligence programming languages, see Bobrow and Raphael [8].

Production systems like OPS2 are often called "pure" production systems, presumably because they contain fewer and more general language constructs than the other production systems. Other languages in this class include PSG [38], one of the earliest production system languages; PSNLST [48], a language intended for general Artificial Intelligence applications; HSP [31], a language in which a large part of a speech recognition program was coded; ACT [2], a language for simulating human cognition; EPS [11], a language much like

PSG; PAS-II [58], another PSG-like language; OPS [21], OPS2's predecessor; and Vere's Relational Production System [57], a language with a more carefully formulated semantics than the others listed here.

Another important class of production systems is the class of deductive or consequent-driven systems. The interpreters for these systems include automatic deduction mechanisms. A production like

A B C --> D E

(A, B, C, D, and E are forms containing constants, variables, etc.) in a pure production system means, "If A, B, and C are instantiated, then instantiate D and E and add them to working memory." In a deductive production system, a production like this means, "To show either D or E to be true, try to show A, B, and C to be true." If the interpreter was asked to show some statement X to be true, it would first try to find a matching statement in working memory. The presence of the working memory element would indicate that X was true. If the interpreter could not find such an element, it would locate a production whose RHS contained a pattern that could be instantiated to X, and then recursively call itself to show all the statements in the production's LHS to be true. Deductive production systems also allow the productions to be executed in the style of pure productions (left to right), though generally a given production cannot be used in both a deductive and a pure style. Perhaps the two best known deductive production systems are Emycin, which has been used for the Mycin [52, 12] and Tieresias [13] systems, and Rita [3, 4], the Rand intelligent terminal agent. Most Emycin productions are executed in the deductive mode; most Rita productions most are executed in the pure mode.

Some production systems have been created just for incorporation into a particular program and have consequently never been made available outside the program. The programs incorporating such production subsystems include Dendral [19], a program for interpreting the data from a mass spectrometer; Hearsay-II [17, 30], a speech recognition system; and AM [29], a program to perform scientific discovery.

Finally, production systems are not unique in using pattern matching to invoke processing. Many of the new Artificial Intelligence languages allow pattern directed invocation of procedures, though in these systems it is not the only way to call a procedure. These languages include Micro-Planner [55, 5], Conniver [32], QA4 [45], Qlisp [50, 42], Plasma [26, 27], and KRL [9, 10].

1.3.2 Relevance of Language Features

A diversity of features exists among the languages listed in the last subsection. Even in the single class of pure production systems, there are different sets of conflict resolution rules, different match and action primitives (though here the differences are generally not great), and three different data element formats. The three data formats are name-attribute-value triples (see Rita), a kind of colored directed graph called a semantic net (ACT), and either OPS2-style lists or a subset of these lists.

Only two of the differences, the data element format and the match primitives, are relevant to the discussion in this thesis. Since the methods described here are concerned with the match, whether the methods could be used for a given language cannot depend on the primitive action types of the language, on the conflict resolution rules used, or on any other feature provided by the language (e.g., the automatic search mechanisms of the deductive production systems). At the end of Chapter 2 there is an argument that the methods described there can be used for most data element formats and for many kinds of match primitives.

1.4 How Big is a Production?

Since this thesis is concerned with the performance of production system interpreters, there are many places in the thesis where it is necessary to discuss the size of a production system or the rate of execution of a production system. The metrics that will be used in these discussions are: for production system size, the number of productions in production memory and the number of data elements in working memory; for the rate of execution, the number of RHS actions performed each second. These metrics are obviously crude, but so are the metrics which are commonly used for conventional programming languages. To measure the size of an Algol program, for example, one often simply counts the lines of code. Any such metric is meaningful only if the person reading the numbers is familiar with the metric. This section provides an introduction to the production system metrics for those readers who may be unfamiliar with them.

The first metric, the number of elements in working memory, can be related directly to Lisp since OPS2 data elements are lists. To have, for example, 300 data elements with an average of 6 list cells per element, is to have 1800 list cells of data.

The other two metrics can also be related to conventional programming languages, though with more difficulty. In the past few years, several programs which were originally written in Lisp and other conventional programming languages have been recoded as production

systems. By comparing the original and production system versions of the programs, it is possible to derive estimates for the amount of processing represented by one RHS action and for the number of lines of code required to equal one production. Appendix I contains the details of comparisons of five programs. In those comparisons, when it was necessary to convert a range into a single number, the number that was least favorable to the production system was chosen. To summarize the results, a production is the equivalent of at least 7 lines of Lisp, and a production system would have to execute about 1000 actions per second to equal the performance of Lisp on a PDP-10, model KL. (A PDP-10, model KL is a medium scale computer, capable of performing more than a million instructions per second when running Lisp.)

1.5 Solutions to the Problem

This section surveys the existing strategies for improving the efficiency of the match.

Most of the strategies involve some kind of indexing scheme. In general, an index is a function from members of one set to members of another set. In Fortran, for example, one can write the expression

```
X(3)
```

which will cause the computer to index from the integer 3 to the floating point number stored in the third element of the vector X. Many indexing schemes involve successive applications of more than index function. In a language that allowed vectors to be stored as elements of other vectors, for example, one could write

```
X(3)(4)
```

which would cause one level of indexing from 3 to a vector, and then another level from 4 to the element stored in the fourth position of that vector. In contrast to these two examples, the index functions used by production system interpreters do not restrict their index sets to integers. The indexing performed in production system interpreters maps from a symbolic expression (typically a list) to elements of some set. The interpreter generally extracts some feature of the expression (perhaps taking the first atom from the list) and then retrieves a precomputed association from that feature to the set.

One of the simplest examples of indexing is found in PSNLST [48]. When an element enters PSNLST's working memory, one feature of the data element is used to associate to all the productions that might match that element. With each production the interpreter keeps a list of the data elements that have associated to it. By comparing the lists the interpreter

determines which productions are most likely to have instantiations, and it tries to instantiate those productions first. Due to the conflict resolution rules of PSNLST, the match can terminate after the first instantiated production is found. Despite the simplicity of the indexing, PSNLST has one of the faster production system interpreters -- in large part because the language was so defined that this indexing scheme would be highly discriminating.

McDermott, Newell, and Moore [33] investigated several indexing schemes. One was a single level scheme similar to that in PSNLST. Two were two-level schemes in which the second level of indexing used a highly discriminating, but computationally expensive combination of features. A fourth scheme they tried was quite different from the others. It used a discrimination net, which on each cycle indexed from the entire contents of working memory to the entire set of productions that might be instantiated. The interpreter using this scheme was less efficient than the other three.

Rieger [44] is investigating a scheme that uses multiple levels of indexing; the number of levels used for each condition element depends on the number of features occurring the element. This system attempts to get further efficiency by partitioning working memory. The user is required to divide working memory into "channels" and to declare which channel each condition element is sensitive to.

Rhyne [43] has proposed an extension the concept of indexing: constructing associations from the actions in the productions' RHSs to the condition elements that might match the data elements added or deleted by the actions. When an element entered or left working memory, the interpreter would retrieve the association stored with the responsible action, rather than compute the index from the element. The scheme seems never to have been implemented, and from its description it appears that it would provide a useful amount of discrimination only if the action elements were composed predominantly of constants.

McCracken [31] has described an interpreter which indexes working memory as well as production memory. Ordinarily, when indexing of productions is used, all the condition elements are indexed. Consequently, when an attempt is made to instantiate a production, the data elements that might match the condition elements are all immediately available. In this interpreter, however, only the first two condition elements of each LHS are indexed. Two other mechanisms are included in the interpreter to make it easy to find the instantiations of the remaining condition elements. The first is using explicit pointers between data elements. In many cases, the match routine can instantiate the remaining condition elements simply by following the pointers from the data elements instantiating the first two condition elements. The second efficiency measure is an index of the elements in working memory. Both the

indices used by HSP are multi-level. The index of the condition elements has from two to five levels, the exact number in each case depending on the constants appearing in the condition elements. The index of working memory has two levels.

The interpreters using indexing all have the property that after the indexing step, another operation has to be performed to determine conclusively whether the productions are instantiated. This operation can be eliminated by using a mechanism that has been known for many years. In 1958 Seifridge [51] presented a general model for the construction of pattern recognizers in his "Pandemonium" machine. The machine consists of a lattice of processing units called "demons". The demons at the bottom of the lattice continuously monitor some area, waiting for a particular event. In a production system this area would be working memory, and the event would be the addition or deletion of a data element possessing a particular feature. When the event occurs, the demon makes a note of that fact and passes the note to the demons directly above it in the lattice. These demons compare the notes they receive from the several demons immediately below them, and when one of these demons finds an interesting pattern in the notes, it sends a note to the demons on the next level. When one of the demons at the top of the lattice finds an interesting pattern in the notes it receives, it takes a more telling action than just sending a note. In a production system interpreter, these topmost demons would make changes to the conflict set. The Acorn system [24] uses a network of demons for parsing a subset of English, a process very similar to the match in a production system interpreter.

In 1974 this writer developed another variation on the Pandemonium model: the Rete Match Algorithm [20]. Since that time the algorithm has been used for four production system languages. Five major variants and numerous minor variants of the algorithm have been tried. A complete description of the latest and most successful form of the algorithm will be found in later chapters.

1.6 Overview of the Thesis

The material presented in Chapters 2, 3, and 6 is perhaps the most important in this thesis. These chapters show how to program a conventional computer to interpret production systems more than an order of magnitude faster than the previous state of the art. (The calculations indicating this are found in Chapter 6.) In addition, Chapter 6 shows how to use microcode to obtain almost another order of magnitude increase in speed, and how to augment the hardware of a conventional computer very slightly and achieve yet another half order of magnitude increase. The material presented in Chapters 4 and 5 is of perhaps less intrinsic interest, but it is needed to support the rest of the work. Chapters 4 and 5 describe the results of analytical and empirical studies of a production system interpreter that uses

the techniques described here.

Chapter 2 introduces the Rete Match Algorithm. It begins by describing some properties of production systems that make it possible to construct efficient interpreters. Then, using examples taken from OPS2 production systems, it develops an algorithm capable of taking advantage of these properties. This algorithm is designed so that it can be implemented on either serial or parallel computers. Despite the algorithm's being motivated by OPS2, it is not specialized for that language; Chapter 2 contains a description of the class of production systems that can use the algorithm. The chapter ends with a description of the algorithm that is as detailed as it can be without being specialized to one language.

Chapter 3 describes the most recent interpreter to use the algorithm, the OPS2 interpreter. This chapter contains descriptions of both the programs in the interpreter and the data operated upon by the programs; enough information is given to allow the reader to reconstruct the OPS2 interpreter. The structure of this interpreter deviates somewhat from the general structure given in Chapter 2. Some of the changes are simply attempts to achieve slightly greater efficiency by taking advantage of the peculiarities of OPS2. One of the changes, however, is of general interest. By taking advantage of the fact that the interpreter was to be run on a serial computer, it was possible both to increase the speed of the interpreter and to decrease the amount of data stored by the interpreter.

Chapter 4 contains an analysis of the OPS2 interpreter algorithm. This chapter determines how the time and space costs of the algorithm depend on the size of the production system. The independent variables considered are the number of productions in production memory and the number of data elements in working memory. The dependent variables are the amount of memory required to store the production system, the amount of memory required for the interpreter's workspace, and the time required to perform the match. This chapter contains expected results as well as best and worst case results. In order to derive the expected results, it was necessary to analyze existing production systems to discover the coding conventions that are generally used. A description of the coding conventions is contained in the chapter. Only the general forms of the functions relating independent and dependent variables are given in this chapter (e.g., it is shown there that in the best case, the time required to perform the match increases with the logarithm of the number of productions in the system).

Chapter 5 presents measurements of the three largest OPS2 production systems. (The production systems contained, respectively, 316 productions, 381 productions, and 1017 productions.) One series of experiments was undertaken to verify and extend the results in Chapter 4. Once again, the two independent variables were the number of productions in

production memory and the number of data elements in working memory, and the three dependent variables were the amount of memory required to store the production system, the amount of memory required for the interpreter's workspace, and the time required to perform the match. Another series of experiments was undertaken to verify that the Rete Match Algorithm could be run efficiently on parallel computers and on computers which use strategies like caching to improve their processor-memory bandwidth.

Chapter 6 considers the effect of specialized hardware on the interpreter's efficiency. This chapter shows that very large increases in speed are possible, and that only minimal hardware modifications are necessary to bring about the increases. The chapter shows how to avoid extensive hardware modifications by adopting appropriate representations for data elements and productions. It contains a detailed description and justification of one possible representation. Calculations made in this chapter show that a slightly modified processor could interpret production systems more than two orders of magnitude faster than current interpreters. This speed increase would be accompanied by an appreciable reduction in the space required to store the productions.

Chapter 7 summarizes the results of the thesis and suggests areas needing further study.

2. The Rete Match Algorithm

This chapter lays the foundations of this thesis by describing the algorithm that will be investigated here, the Rete Match Algorithm. The first four sections try to motivate the algorithm. These sections explain how the algorithm tries to get efficiency, and using example productions drawn from the MKYBAN production system, they show exactly what the algorithm has to do. The next section contains a concise description of the algorithm developed in the first four sections. The chapter concludes with a discussion of the class of production systems that can use the Rete Match Algorithm.

2.1 Introduction to the Algorithm

This section introduces the Rete Match Algorithm, describing the properties of production systems that the algorithm tries to exploit, and explaining how the basic organization of the algorithm helps to exploit them. These production system properties are called temporal redundancy and structural similarity.

2.1.1 Temporal Redundancy

In most production systems, working memory changes rather slowly from cycle to cycle. Working memory sizes typically fall in the range of fifty to five hundred elements. Firing a production will typically change two to five of these elements. This fact is important in the match algorithm design because, with most of the data elements unchanged, most of the information used by the match for one cycle could be used by the match for the next cycle. The two to five changes performed by the production that fires may result in a few productions being instantiated and in a few others loosing their instantiations, but most productions will neither gain nor loose any instantiations. Moreover, any production that becomes instantiated was probably close to being instantiated on the previous cycle -- perhaps having instantiations for all but one of its condition elements. The match routine can take advantage of this by saving from one cycle to the next an indication of which productions are instantiated and which condition elements of the other productions are instantiated, incrementally updating the information to reflect the changed state of working memory. If it does so, the effort required to perform the match will depend more on the small number of elements changed than on the much larger total number of elements in working memory.

2.1.2 Structural Similarity

The LHSs of a production system usually exhibit a substantial amount of similarity. Identical condition elements often occur in different productions. Different condition elements

often have identical lengths or contain identical constants. MB11's condition element, for example, is identical to MB12's first condition element. MB12's second and third condition elements both have three subelements, and each has **Monkey** as its first subelement.

```
MB11    ( (Want (Monkey Near =P))
          -->
          (Want (Monkey On Floor)) )

MB12    ( (Want (Monkey Near =P)) (Monkey On Floor)
          (Monkey Near =C & #P)
          -->
          (<WRITE> "The monkey walks from" =C "to" =P)
          (<DELETE> (Want (Monkey Near =P)) )
          (<DELETE> (Monkey Near =C) )
          (Monkey Near =P) )
```

If the match routine can recognize the occurrence of identical features, it can avoid making the same test multiple times. After determining, for example, that MB11 had no legal instantiations, it would not waste time trying to instantiate MB12.

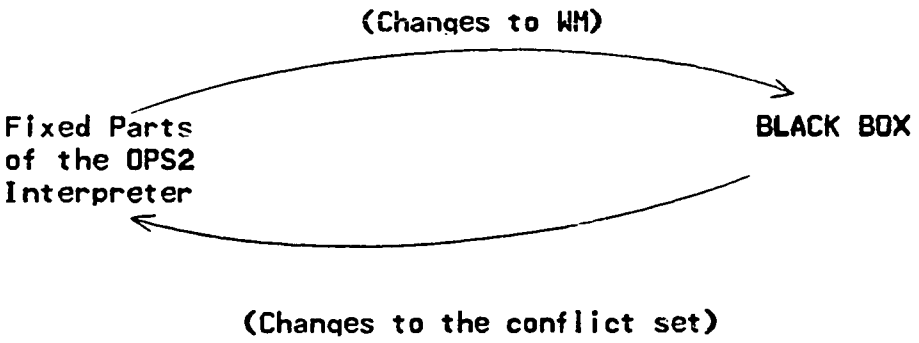
2.1.3 Compiling the LHSs

In order to take advantage of structural similarity, an interpreter using the Rete Match Algorithm must process the LHSs of a production system before beginning execution of the system. The preprocessing step is necessary because it is in this step that the similarities are discovered.

The preprocessing step is a compilation that builds a part of the interpreter. An interpreter using the Rete Match Algorithm is incomplete until a set of productions is compiled. The interpreter contains programs to perform conflict resolution, to execute the productions' RHSs, and to provide various utility functions like tracing; but it contains no programs to interpret the productions' LHSs and perform the match. From the production system to be executed, the LHS compiler builds a program to perform the match for that one production system. Since the program constructed by the LHS compiler contains all the information that is necessary to perform the match, the LHSs may be discarded after it is built. Most of this chapter is devoted to describing the structure of these programs and the way control is passed between their parts.

2.1.4 Overview of the Rete Match Algorithm

Perhaps the most unusual feature of a match routine based on the Rete Match Algorithm is that it never examines working memory. Instead, it monitors the changes made to working memory and maintains internally information which is equivalent to that in working memory. At the beginning of a cycle the match routine notes the changes made to working memory on the previous cycle. From this information the match routine computes whether any changes need to be made to the conflict set. If there are changes to be made, it sends a list of the changes to the fixed part of the interpreter, where the conflict set is maintained. With the state of working memory constantly changing, one production cannot remain satisfied indefinitely, so the changes to the conflict set include removing old instantiations as well as inserting new ones. To the rest of the interpreter, the match routine looks like a black box with one input and one output:



2.1.5 The Kinds of Working Memory Changes Supported

In order to keep this chapter from becoming longer than necessary, it will be assumed here that only two kinds of changes are made to working memory: adding an element and deleting an element. These two were chosen because they comprise a minimal logically complete set. Any of the other action types provided by a production system language can be simulated by combinations of these two. Element modification, for example, which is provided in many languages, can be simulated by one add and one delete. The modify action can appear in the RHSs of the productions just as it would if it were supported directly by the match. When a modify is executed, a routine in the interpreter (invisible to the user) intercepts it and converts it into a delete of the old form of the element and an add of the new form.

Perhaps it should be noted that this assumption is not made because the Rete Match Algorithm is unable to handle other action types. The next chapter describes a version that handles three action types.

2.2 Non-negated Condition Elements

Describing the algorithm will be easier if negated condition elements (i.e., condition elements that must fail to match data elements) are delayed to a later section. This section considers all other language features, showing how they can be compiled into the black box described above.

2.2.1 Productions with one Condition Element

The match programs built by the LHS compiler are Pandemonium-like networks of simple feature recognizers because this organization helps in taking advantage of structural similarity. As will be shown below, it is particularly easy to locate common structure in these networks and eliminate it. The details of these programs will be illustrated by compiling several productions. The simplest productions, those with only one condition element, are considered first.

Production MB11 from the MKYBAN production system is typical of these.

```
MB11    ( (Want (Monkey Near =P))
          -->
          (Want (Monkey On Floor)) )
```

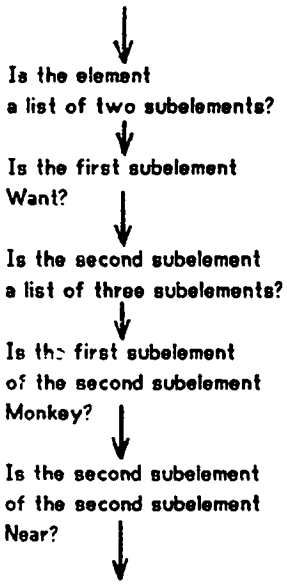
The OPS2 language definition specifies how this production's LHS is to be interpreted. The LHS can be instantiated by any element in working memory that

1. Has two subelements.
2. Has the constant **Want** as its first subelement.
3. Has a list of three elements as its second subelement.
4. Has the constant **Monkey** as the first subelement of the second subelement.
5. Has the constant **Near** as the second subelement of the second subelement.

No mention is made of the variable =P because it can be instantiated by anything that can occur in the OPS2 working memory.

The nodes in the networks test for the occurrence in data elements of features like these. Each of the nodes has a single edge leading into it and one or more edges leading out. Data elements are sent to the node over the one incoming edge. Upon the arrival of a data element, the node is activated to test for the presence of a particular feature. If the test

succeeds, the data element is sent out along every edge leaving the node. If the test fails, the data element is consumed by the node. Since MB11 has five features to test, there are five of these nodes in its network. These nodes are arranged in a linear sequence; the output of one node is the input of the next:

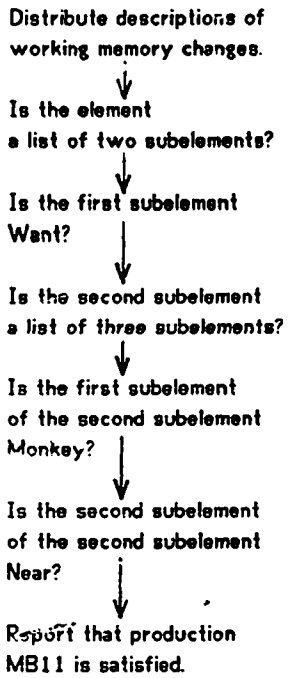


No input for the first edge or output for the last edge is shown because the nodes at the ends of these edges do not test data elements. They simply connect this network into the black box.

The node that supplies the inputs for the first node above might be called the input bus of the black box. This node accepts descriptions of the changes being made to working memory and distributes the information to every first node in every condition element's network. The network contains only one input bus node regardless of the number of productions in the system.

The last edge in the above sequence connects to a node which is concerned with maintaining the conflict set. When this conflict set node receives a data element, it sends the data element and the name of the production to the routines that maintain the conflict set. Since nodes of this kind have to know the names of the productions, a separate node has to be built for each production in the system.

The complete network for MB11 is



2.2.2 The Data Processed by the Nodes

The packets sent between nodes are called tokens. Every token has two components, a tag part and a data part. The data part of the token holds either a working memory element or a list of working memory elements. The uses of the tag part cannot be fully explained until more node types have been described. In nodes of the type shown above, the tag part simply holds an indication of the kind of change that was just made to working memory (with the data part holding the affected data element). If only two action types are to be supported by the match, only two tags are needed, VALID for elements just added to working memory and INVALID for elements just deleted from the memory. In this chapter tokens will be represented as ordered pairs. The first part of the pair is the tag, and the second part is the data part. The tokens that would be processed during execution of the MKYBAN production system include the following.

```
<VALID, (Want (Monkey Holds Bananas))>
<VALID, (Want (Monkey On Floor))>
<INVALID, (Want (Monkey On Floor))>
```

2.2.3 Processing in the Network for MB11

When a problem like task T1 is started, several data elements are forced into working memory. The elements in the case of T1 include **(Want (Monkey Holds Bananas))** and

(Monkey Near (5 7)). When these enter working memory, the interpreter builds tokens with VALID tags, and the first node in the network passes copies of the tokens to its immediate successors. One of its successors is the node that performs the length test for MB11.

```
Is the element
a list of two subelements?
```

When this node receives the token

```
<VALID, (Want (Monkey Holds Bananas))>
```

it tests the data part, finds that the length of the data part is two, and sends the token to its successor.

```
Is the first subelement
Want?
```

This node tests the first subelement of the data part, and since it is **Want**, it passes the token to its successor.

```
Is the second subelement
a list of three subelements?
```

This node tests the length of the second subelement and passes the token to its successor.

```
Is the first subelement
of the second subelement
Monkey?
```

This node tests the first subelement of the second subelement, finds that it is **Monkey** and passes the token along to its successor.

```
Is the second subelement
of the second subelement
Near?
```

This node rejects the token; it tests the second subelement of the second subelement and

finds **Holds** rather than **Near**. It sends nothing to its successor, and processing of this token in MB11's network concludes. The rest of the tokens processed when T1 starts are rejected faster than the first one was. The token

<VALID, (Monkey Near (5 7))>

for example is rejected after the first test is applied.

Elsewhere in the network, though, other nodes accept these tokens, and the conflict set has instantiations added to it. These instantiations execute, resulting in new elements being added to working memory and old elements being deleted. None of the tokens created to represent these changes pass completely through MB11's network until **(Want (Monkey Near (2 2)))** is added. The token

<VALID, (Want (Monkey Near (2 2)))>

passes every test in MB11's network and causes the last node to add an instantiation of MB11 to the conflict set. Later **(Want (Monkey Near (2 2)))** is deleted, causing token

<INVALID, (Want (Monkey Near (2 2)))>

to be created and processed. MB11's network passes this token also. The arrival of the INVALID token at the last node causes the instantiation of MB11 to be removed from the conflict set.

2.2.4 Productions with Two Condition Elements

A LHS with two condition elements compiles into a network containing two linear sequences of nodes (one for each condition element) plus one node to combine the tokens passed by these two sequences. This section is primarily concerned with describing this last node. For simplicity's sake, it begins by considering LHSs in which the condition elements share no variables.

Production MB18 is typical of the productions considered by this section.

```
MB18    ( (Want (EmptyHanded Monkey)) (Monkey Holds =X)
          -->
          (<WRITE> "The monkey drops the" =X)
          (<DELETE> (Want (EmptyHanded Monkey)) )
          (<DELETE> (Monkey Holds =X) ) )
```

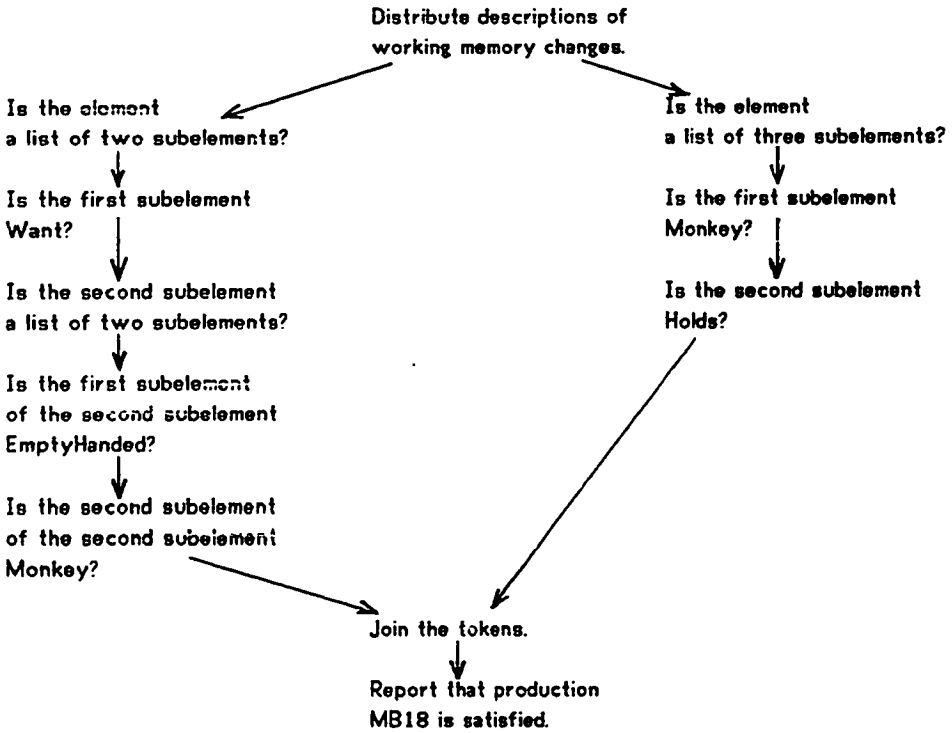
The first condition element is instantiated any data element that

1. Has two subelements.
2. Has the constant **Want** as its first subelement.
3. Has a list of two subelements as its second subelement.
4. Has the constant **EmptyHanded** as the first subelement of its second subelement.
5. Has the constant **Monkey** as the second subelement of its second subelement.

The second condition element is instantiated by any data element that

1. Has three subelements.
2. Has the constant **Monkey** as its first subelement.
3. Has the constant **Holds** as its second subelement.

As in the network for MB11, each of these features compiles into a separate node.



The responsibility of the node with two inputs is to join pairs of data elements -- one element from the left branch of the network and one from the right -- into lists of two elements. Since the elements do not generally arrive simultaneously, if it is to perform this function, it must save state from one activation to the next. Suppose, for example, that the

MKYBAN production system is running, but that no element has yet entered working memory that matches either of the two condition elements. Then the two-input node in MB18's network has not been activated, and it is maintaining no state. At some time during the run, the element **(Monkey Holds Ladder)** enters working memory. The token

<VALID, (Monkey Holds Ladder)>

is processed, and manages to pass all the nodes in the right branch of this network. The token reaches the right input of the two-input node and stops. Since the node has received no tokens from the other input, it can build no lists. To prepare for the possible arrival of a token on the left input at a later time, it stores in an internal memory a note to itself, "I received the token **<VALID, (Monkey Holds Bananas)>** over my right input." The node does no more on this activation.

The processing of MKYBAN continues, and eventually the goal **(Want (EmptyHanded Monkey))** enters working memory. The token

<VALID, (Want (EmptyHanded Monkey))>

is processed by the network, passing through the left branch of MB18's network, and arriving finally at the left input of the two-input node. The node examines its memory, finds the note it made earlier, and builds the token

<VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Ladder)>.

This token is sent to the last node where it causes MB18's new instantiation to be added to the conflict set. Before the two-input node terminates processing, it adds another note to its internal memory, "I received the token **<VALID, (Want (EmptyHanded Monkey))>** over my left input." This will be needed if it receives another token over its right input.

2.2.5 Deleting Elements from Working Memory

When elements are deleted from working memory, the match routine must update both the conflict set and those of its node memories that have stored the element. The examples of processing in MB11's network (section 2.2.3) showed how the tags are used in the updating of the conflict set. This section shows how they are used in the updating of the node memories. In particular, it shows how the use of tagged data allows updating the node memories without examining every node in the system.

Consider the example in the last section again. After MB18's two-input node finishes

processing the second token, it has stored

<VALID, (Want (EmptyHanded Monkey))>

received from the left and

<VALID, (Monkey Holds Ladder)>

from the right. MB18 probably fires soon after the second activation of this node. When it does, it deletes **(Want (EmptyHanded Monkey))** and **(Monkey Holds Ladder)**. Suppose the resulting tokens are processed in this order. Then

<INVALID, (Want (EmptyHanded Monkey))>

passes through the left branch of the network and arrives at the left input of the two-input node. The INVALID tag of this token causes the two-input node to delete the note it made before, "I have received the token **<VALID, (Want (EmptyHanded Monkey))>**". The two-input node then builds a new token much like the one it built when the VALID token arrived; it takes the element **(Monkey Holds Ladder)** from its right input memory and joins it to the data element from the token that just arrived to produce the token

<INVALID, (Want (EmptyHanded Monkey)) (Monkey Holds Ladder)>.

This token is sent to the last node where it causes the instantiation of MB18 to be removed from the conflict set.

One might ask why the two-input node tagged the token it built with the tag of the token that just arrived (INVALID) rather than the tag of the token in its memory (VALID). To answer this it is necessary to understand the reason the token was built. The token

<INVALID, (Want (EmptyHanded Monkey))>

could not have arrived at the left input of the node unless the token

<VALID, (Want (EmptyHanded Monkey))>

had arrived earlier, for a data element cannot be deleted from working memory unless it is present there. Since there was a note in the node's memory that the token

<VALID, (Monkey Holds Ladder)>

had arrived on the right input, it had to be true that, at some time in the past, the node had produced as output the token

<VALID, (Want (EmptyHanded Monkey)) (Monkey Holds Ladder)>.

This node was built because, at the time of its building, it was necessary to add its data part to the memories succeeding this node. But after the arrival of

<INVALID, (Want (EmptyHanded Monkey))>

it was no longer correct to keep the data part in the memories. The node then had to create

<INVALID, (Want (EmptyHanded Monkey)) (Monkey Holds Ladder)>

to inform its successors of the changed situation.

After processing the token resulting from the deletion of (Want (EmptyHanded Monkey)), the match processes the one resulting from the deletion of (Monkey Holds Ladder):

<INVALID, (Monkey Holds Ladder)>.

This token passes down the right branch of the network and arrives at the right input of the two-input node. The node is activated to search through its node memory and delete the note, "I have received the token <VALID, (Monkey Holds Ladder)> over my right input." Since the node has already deleted the note about the one VALID token that arrived from its left, it produces no outputs.

In the network for the entire MKYBAN production system there are many two-input nodes (every one of which maintains state like this one) yet only a few were activated on this cycle. Because the INVALID tokens are subject to the same tests as the VALID tokens, only the nodes whose memories needed to be changed were activated.

2.2.6 How the Two-input Nodes Handle the Tags

To summarize the previous sections, the two-input nodes observe the following three rules for handling tags.

1. When a token arrives that is tagged VALID, store that token in the internal memory.

2. When a token arrives that is tagged INVALID, delete from the internal memory a

token with an identical data part.

3. When an output token is created, copy the tag of the token that just arrived.

Some two-input nodes test the data elements before building tokens to send to their successors (see section 2.2.8). The two-input nodes ignore the tags while they are performing these tests.

2.2.7 The Internal Memories of Two-input Nodes

The notes in the examples above ("I have received the token <VALID, (Want (EmptyHanded Monkey))> on my right input.") are longer than they need be. They show the two-input nodes storing three pieces of information about each token: the tag of the token, the data part of the token, and an indication of whether it arrived on its left or right input. Since, as the rules for handling tags show, only tokens tagged VALID are stored, the tags can be dispensed with. The indication of whether the token arrived on the left or right input cannot be dispensed with, but it can be factored out of the tokens to save space. Each node can have two memories rather than one. Tokens arriving from the left can be stored in one memory, and tokens arriving from the right in the other. Thus only the data parts of the tokens need to be stored explicitly.

2.2.8 Variables Occurring in More Than One Condition Element

This section begins the discussion of networks for productions in which variables occur more than once. Production MB15, with the variable =O occurring in both its condition elements, is typical.

```
MB15    ( (Want (Monkey On =O)) (=O Near =X)
          -->
          (Want (Monkey Near =X)) )
```

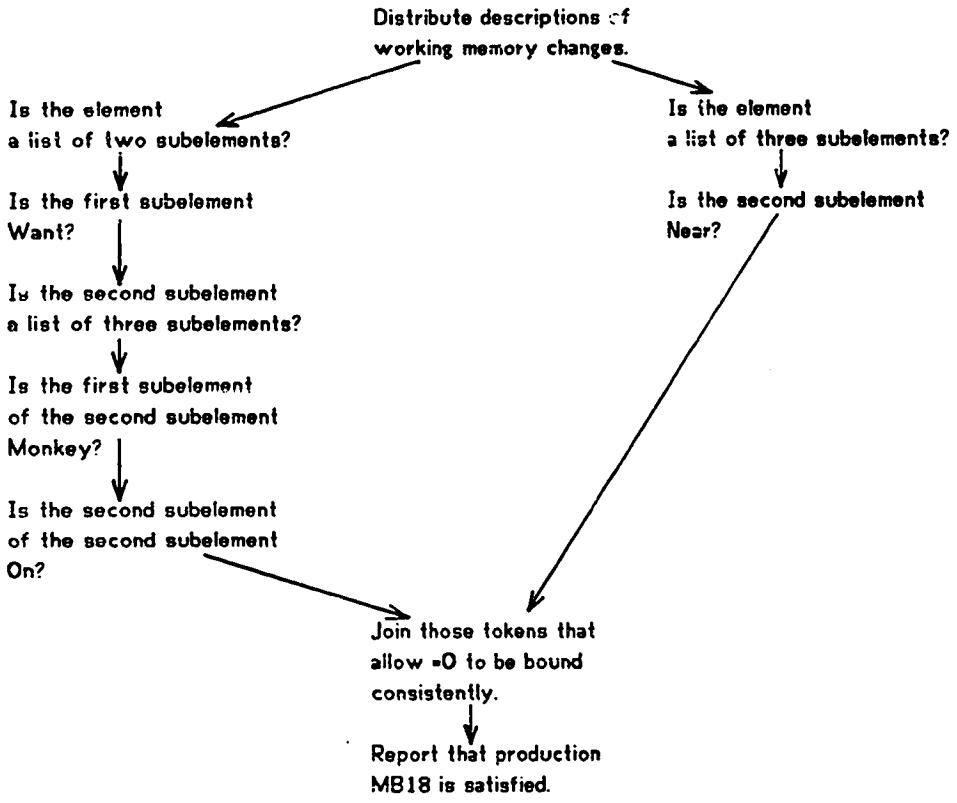
Instantiating this production involves the testing of both isolated data elements and pairs of data elements. The tests of isolated elements are made, as in the previous examples, to test features like the lengths of lists and the constants in particular positions. These tests are used for example to locate for the second condition element a data element that

1. Has three condition elements.

2. Has the constant Near as its second subelement.

The tests of pairs of data elements are made to insure that both occurrences of the variable =O are bound to the same data subelement.

In the Rete Match Algorithm, the two-input nodes perform the tests for consistency of variable binding. The network for MB15, including the test for =0 is



To see how this two-input node would work, suppose that the three data elements

```
(Monkey Near (5 7))
(Want (Monkey Near Ladder))
(Ladder Near (2 2))
```

enter working memory in this order. Suppose also that nothing else in working memory can pass either set of one-input nodes for this production. When the first of the data elements enters working memory, its token passes through the right branch of the network, and the data part of the token is stored into the two-input node's right memory. Since the node has received nothing on its left input, it produces no output. When the second element enters working memory, its token passes through the left branch of the network. The two-input node stores the data part of the token in its left input memory and then examines its right input memory. It finds **(Monkey Near (5 7))** there, but since joining these two elements would cause =0 to be bound simultaneously to both **Ladder** and **Monkey**, the two-input node still produces no output. Finally **(Ladder Near (2 2))** enters working memory. When the token for this element arrives at the right input of the two-input node, the node stores away its data part and then examines the left input memory, finding

(Want (Monkey On Ladder)). The variable =O can be bound consistently, so the node outputs the token

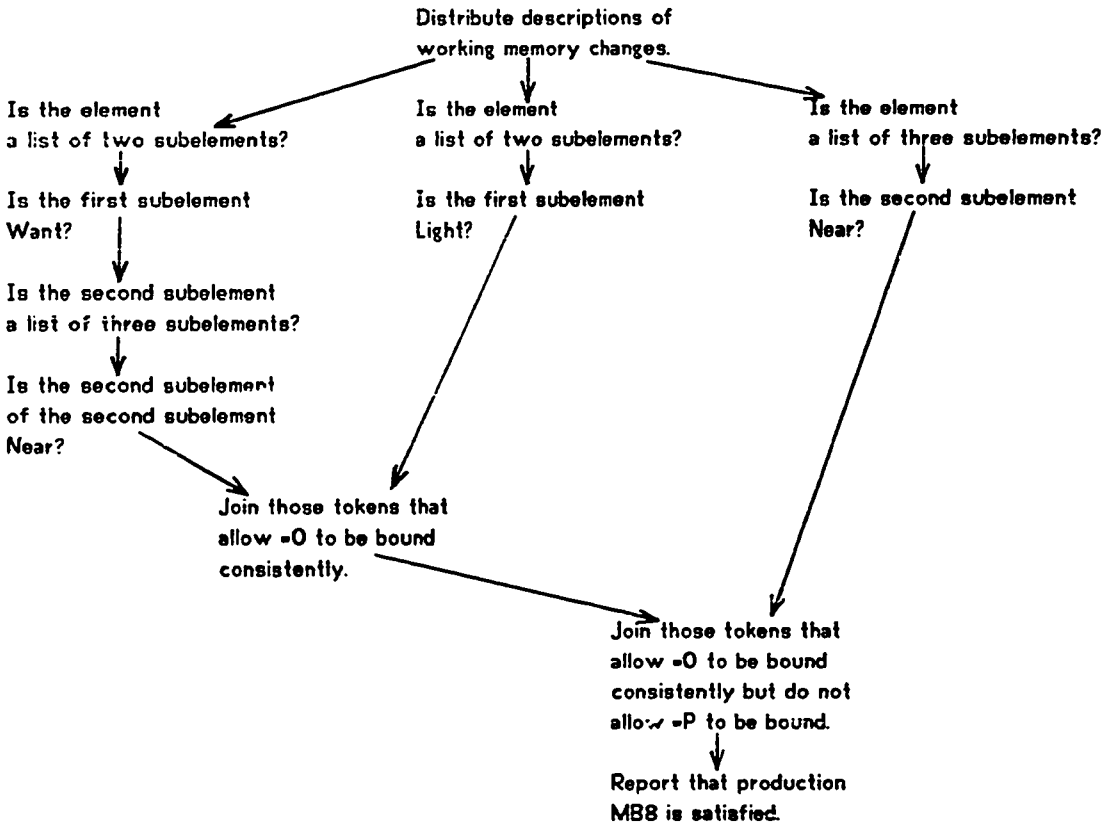
<VALID, (Want (Monkey On Ladder))(Ladder Near (2 2))>.

2.2.9 More Complex Productions

Many productions are more complex than MB15, containing either more condition elements or more variables. Production MB8, for example, contains three condition elements and two variables. One of the variables occurs in all three condition elements, and two of the condition elements contain occurrences of both variables.

```
MB8     ( (Want (=O Near =P)) (Light =O) (=O Near #P)
          -->
          (Want (Monkey Holds =O)) )
```

Having more condition elements makes it necessary to have more two-input nodes in the network; in general, a LHS with K condition elements compiles into a network with K-1 two-input nodes. Having more than one variable in some condition elements makes it necessary to have more than one test in some two-input nodes. The network for MB8, for example, is



For an example of the processing in this network, consider problem T2. The elements forced into working memory when T2 is started include **(Ladder Near (2 2))** and **(Light Ladder)**. The tokens representing these two elements are processed, passing through the rightmost and the center branches of the network, respectively. The token for **(Light Ladder)** is stored in the right input memory of the first two-input node. The token for **(Ladder Near (2 2))** is stored in the right input memory of the second two-input node. Later, the element **(Want (Ladder Near (8 2))** enters working memory, and its token passes down the left branch of MB8's network. When the first two-input node receives the token, it builds

```
<VALID, (Want (Ladder Near (8 2))) (Light Ladder)>
```

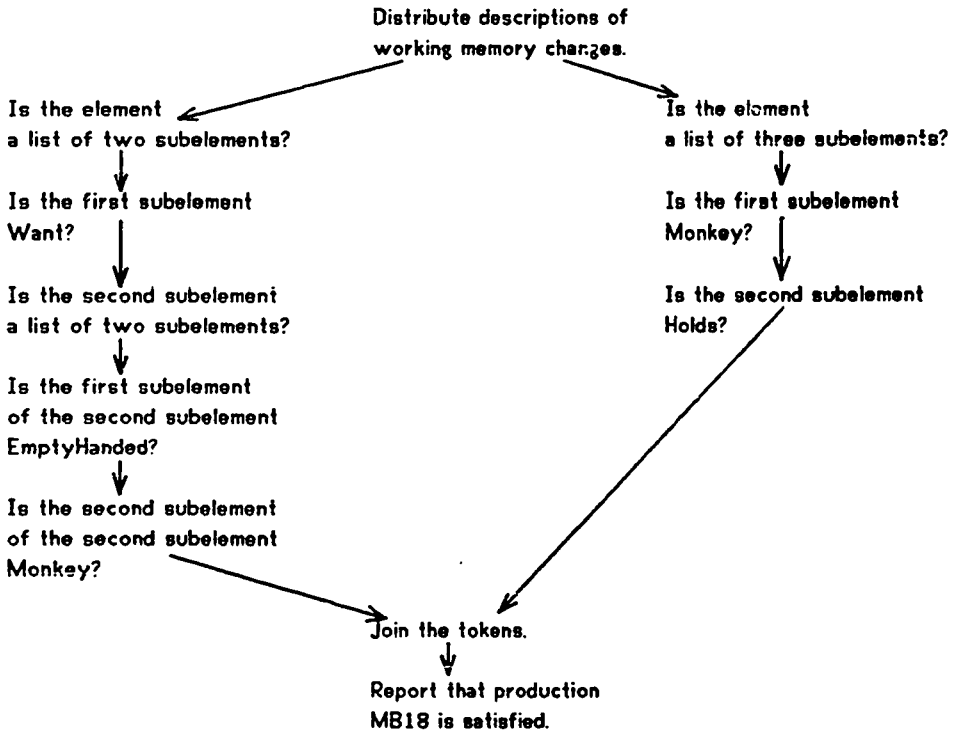
and sends it to the other two-input node. That node builds

```
<VALID, (Want (Ladder Near (8 2)))
                (Light Ladder)(Ladder Near (2 2))>
```

and sends it to the last node. The last node adds the instantiation of MB8 to the conflict set.

2.2.10 Producing Multiple Output Tokens

In the examples presented above, the arrival of a token at a two-input node resulted in at most one token being output. In practice a node often produces more than one output token. Consider the network for production MB18 again.



If the following elements entered an empty working memory, the two-input node would produce no outputs, but it would make one entry in its right input memory for each data element.

```
(Monkey Holds Ladder)
(Monkey Holds Orange)
(Monkey Holds Glass)
(Monkey Holds Box)
```

If (Want (EmptyHanded Monkey)) entered working memory later, its token would reach the left input of the two-input node, and cause processing which would result in four tokens being output.

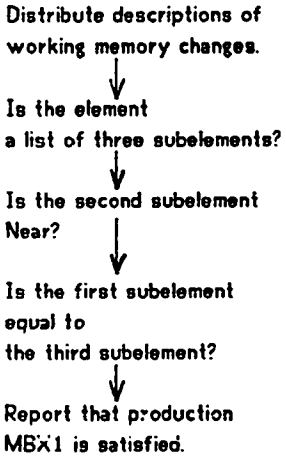
```
<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Ladder)>
<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Orange)>
<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Glass)>
<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Box)>
```

2.2.11 Variables Occurring Multiple Times in One Condition Element

Sometimes a variable will occur more than once in a condition element. No examples of such variables are to be found in MKYBAN, but they might be needed in a more general system. If the relation Near were allowed to hold between two objects, and if the system incorporated rules about transitivity of nearness (if A is near B and B is near C then A is near C) it might be necessary to incorporate rules to recognize incorrect applications of transitivity. It would be easy to make the mistake of asserting that some object is near itself. The production to correct this could simply erase the data element:

```
MBX1    ( (=X Near =X) --> (<DELETE> (=X Near =X)) ).
```

Since there are no two-input nodes in the network for MBX1, the bindings for =X must be tested for consistency by a one-input node.



Even if a production contains more than one condition element -- and therefore has two-input nodes in its network which could test all the variable bindings -- it is better to test as many as possible in the one-input nodes. Putting the tests before the two-input nodes reduces the number of tokens that reach them and have to be stored in their input memories.

2.3 Negated Condition Elements

A new node type is needed to compile LHSs containing negated condition elements. This node has two inputs, but it is quite different from the two-input nodes seen earlier in this chapter. This section describes this new two-input node.

2.3.1 The <NOT> Node

The examples in this section use production MB19.

```
MB19    ( (Want (EmptyHanded =X)) - (=X Holds =)
          -->
          (<DELETE> (Want (EmptyHanded =X)) ) )
```

Since this LHS is interpreted differently from the other LHSs discussed in this chapter, it is worthwhile to state exactly what the match routine must do to instantiate MB19. First the match must find individual data elements that instantiate the two condition elements. The first condition element can be instantiated by any data element that

1. Has two subelements.
2. Has the constant **Want** as its first subelement.
3. Has a list of two subelements as its second subelement.

4. Has the constant **EmptyHanded** as the first subelement of its second subelement.

The second condition element can be instantiated by any data element that

1. Has three subelements.

2. Has the constant **Holds** as its second subelement.

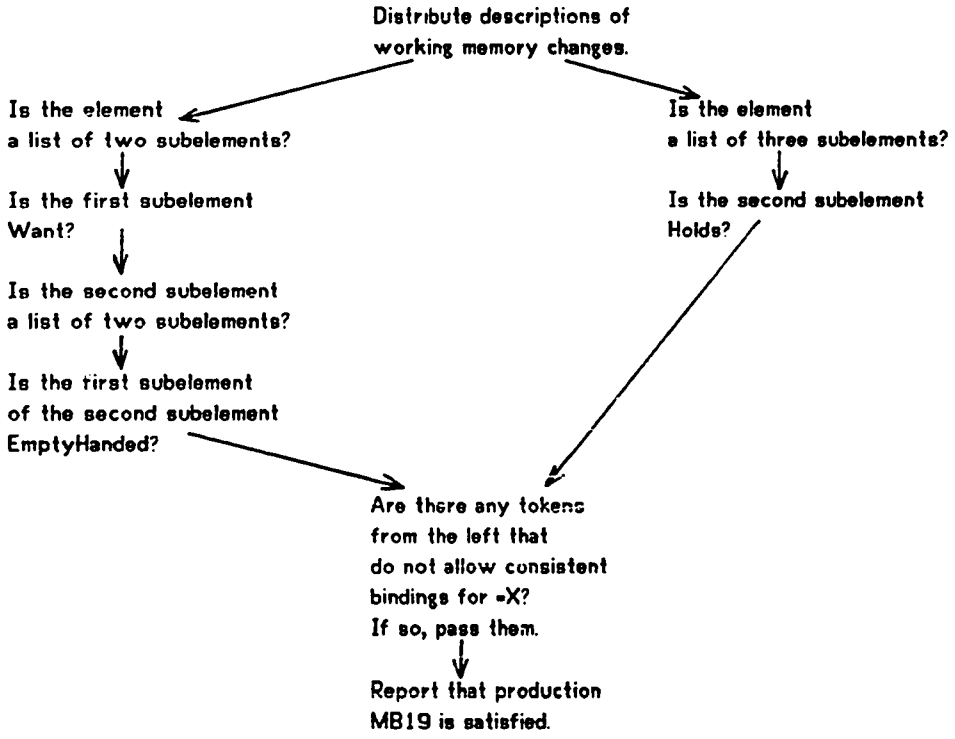
The actions taken next depend on whether data elements are found to instantiate one, both, or neither of the condition elements.

- If there are instantiations for the first condition element and none for the second condition element, the LHS is satisfied. Hence nothing more need be done in this case.

- If there is no data element instantiating the first condition element, the LHS is unsatisfied; whether there are instantiations of the second condition element is unimportant. So in this case also nothing more need be done.

- If there are instantiations of both condition elements, the match must test the bindings to =X. Each possible instantiation of the first condition element is tested against every possible instantiation of the second. If there is an instantiation of the first that has a consistent binding with none of the instantiations of the second, the LHS is satisfied. Otherwise it is not.

The tests for the individual subelement features compile into the usual one-input nodes. The new two-input node makes the tests for consistency in variable binding. The network for MB19 is



The minus sign is used in OPS2 as a kind of a macro to be expanded by the compiler. Production MB19 is interpreted by the compiler as if it were written

```
MB19    ( (Want (EmptyHanded =X)) (<NOT> (=X Holds =) )
          -->
          (<DELETE> (Want (EmptyHanded =X)) ) )
```

The advantage of the longer form is that, with the arguments delimited by parentheses, any number of them may be given to <NOT>. The same kind of two-input node is used to join a <NOT> to the surrounding non-negated condition elements, regardless of the number arguments the <NOT> has. This two-input node will be called a "<NOT> node".

2.3.2 Memories at the <NOT> Node

One of the few similarities in the two kinds of two-input nodes is that they both maintain internal memories. Both store information in their memories when tokens tagged VALID arrive, and both delete the information if the same token tagged INVALID later arrives. Both have one memory for tokens arriving on the left input and another for tokens arriving on the right, and both read from the memory of one input when a token arrives on the other input. Despite these similarities, however, the two node types differ somewhat in their use of the memories, as the following sections will show.

2.3.3 New INVALID Tokens Arriving on the Right

Describing the processing of the <NOT> node is simpler if individual treatment is given to each of the various situations the node has to cope with. The arrival on the right of a token tagged INVALID is potentially the most difficult of the situations. Because the processing the node performs in the other cases is determined partly by the need to make handling this case easier, this case is considered first.

An example will show the problem that must be solved to handle this case correctly. Suppose MKYBAN is working with two monkeys at the same time. The production system has been running for a number of cycles, and the changes made to working memory have caused the <NOT> node to receive the tokens

```
<VALID, (Monkey1 Holds Ladder)>
<VALID, (Monkey1 Holds Orange)>
<VALID, (Monkey2 Holds Orange)>
```

over its right input and the token

<VALID, (Want (EmptyHanded Monkey1))>

over its left input. Production MB19 is not instantiated. If the token

<INVALID, (Monkey1 Holds Orange)>

arrives over the right input, MB19 is still not instantiated. But if

<INVALID, (Monkey1 Holds Ladder)>

arrives next, MB19 is instantiated. The remaining entry in the right memory -- for (Monkey2 Holds Bananas) -- does not prevent the instantiation because =X cannot be bound to Monkey1 and Monkey2 simultaneously. If the two INVALID tokens had arrived in the opposite order, MB19 would have been instantiated only after the arrival of

<INVALID, (Monkey Holds Orange)>.

Regardless of the order in which the two INVALID tokens arrive, the node should send out nothing after the arrival of the first, but after the arrival of the second it should send out the token

<VALID, (Want (EmptyHanded Monkey))>.

The problem to be solved in the design of the <NOT> node is how to determine when to send out tokens like this without excessive effort.

The solution adopted in the Rete Match Algorithm is to store a count along with every entry in the left memory. This count indicates how many entries in the right memory allow variables to be bound consistently. When an INVALID token arrives on the right input and causes an element to be deleted from the right memory, the variable binding tests are repeated for every element in the left memory. When one is found that has consistent variable bindings, the count is decremented by one. If a count becomes zero, the node builds a token tagged VALID for the corresponding entry in the left memory. In the example above, the count for (Want (EmptyHanded Monkey1)) was initially two. When the first token arrived from the right, the count was decremented to one. When the next token arrived, it was decremented to zero, resulting in the sending out of

<VALID, (Want (EmptyHanded Monkey1))>.

One reason for choosing this solution to the problem is that it keeps reasonably small the amount of state to be stored in the <NOT> nodes. The right memories store the same information as the right memories of the other kind of two-input nodes: the data parts of the VALID tokens. The left memories store an integer along with each data part.

2.3.4 New VALID Tokens Arriving on the Right

The events resulting from the arrival of a VALID token on the right input of a <NOT> node *are the inverse of those resulting from the arrival of an* INVALID token. First the node stores the data part of the token in its right memory. Then it examines every entry in its left memory, incrementing the count of those that allow consistent variable bindings. When it increments a count from zero to one, it builds a token for the entry and sends the token to its successors. The new token is tagged INVALID.

To see how this works, consider the example from the last section again. Suppose the VALID tokens had arrived at the two-input node in this order: first

```
<VALID, (Want (EmptyHanded Monkey1))>
```

on the left input and then in order

```
<VALID, (Monkey2 Holds Orange)>
<VALID, (Monkey1 Holds Ladder)>
<VALID, (Monkey1 Holds Orange)>
```

on the right input. When the node received the token from the left, it stored the data part in its left memory with a count of zero and then passed the token to its successor. (See the next section.) When it received the first token from the right, it stored the data part in its right memory and then tested the elements it had stored in its left memory. Since the element there, `(Want (EmptyHanded Monkey1))`, did not allow consistent variable bindings, it left the count unchanged and produced no output tokens. When the next token arrived, the node stored its data part and then examined the left memory again. This time the node found the variable could be bound consistently, so it increased the count from zero to one. This change in count caused it to output the token

```
<INVALID, (Want (EmptyHanded Monkey1))>.
```

Finally the last element arrived on the right. The node stored the data part and examined its left memory a third time. It found the variable could be bound consistently, and incremented

the count again. Since it changed the count from one to two this time, however, the node produced no output.

2.3.5 Tokens Arriving From the Left

A <NOT> node treats all tokens arriving from the left similarly. The only difference is that the node makes an entry in its left memory when the token is tagged VALID and deletes an entry when it is tagged INVALID. Because of the similarity, the two kinds of tokens are considered together in this section.

When there is nothing in the right memory, the <NOT> node passes to its successors every token it receives on its left input. Suppose that the right memory of MB19's <NOT> node was empty when

```
<VALID, (Want (EmptyHanded Monkey))>
```

arrived on its left input. The node would store the data part of the token in its left memory with a count of zero. It would then send the token (unchanged) to its successor. If

```
<INVALID, (Want (EmptyHanded Monkey))>
```

later arrived on the left, the node would delete the information it stored before and send this new token to its successor.

When there are elements stored in the right memory, the node must examine them before it can determine what to do with a token arriving on the left. If the node finds an element in the right memory that allows the variables to be bound consistently, the node sends out nothing. If it finds no elements that allow consistent variable bindings, the node passes the token to its successors. If the token is tagged INVALID, the node then deletes an entry from its left memory. If the token is tagged VALID, the node stores it in its left memory. In deciding whether to produce any outputs, the node would have counted the number of entries in the right memory that allowed consistent variable bindings. It stores this count along with the data part of the token. Thus if

```
<VALID, (Want (EmptyHanded Monkey1))>
```

arrived on the left input of MB19's <NOT> node after

```
<VALID, (Monkey1 Holds Ladder)>
<VALID, (Monkey1 Holds Orange)>
<VALID, (Monkey2 Holds Orange)>
```

had arrived on the right, the node would produce no output, and it would store **(Want (EmptyHanded Monkey1))** in its left memory along with a count of two. If

<INVALID, (Want (EmptyHanded Monkey1))>

arrived next, it would delete the entry it just made, and again produce no output.

2.3.6 How the <NOT> Nodes Handle the Tags

The rules for handling tags given in section 2.2.6 are not used by the <NOT> nodes. The rules for these nodes are:

1. When a token arrives that is tagged VALID, store that token in the internal memory.

2. When a token arrives that is tagged INVALID, delete from the internal memory a token with an identical data part.

3. When an output token is created as the result of processing a token that arrived from the left, copy the tag of the token that just arrived.

4. When an output token is created as the result of processing a token that arrived from the right, invert the tag of the token that just arrived and make that the tag of the new token.

In addition, the tags of the tokens arriving on the right input are used to determine whether to increment or decrement the counts in the left memory.

2.4 Efficiency Issues

This section is concerned with the cost of using the Rete Match Algorithm. The first two subsections explain how the algorithm takes advantage of temporal redundancy and structural similarity. The last subsection shows that the algorithm is capable of being executed in parallel. Parallel execution is an efficiency issue because an algorithm which cannot be executed in parallel cannot make efficient use of modern hardware. With present day hardware technology, the least expensive way to build a powerful computer is to build a highly parallel computer.

2.4.1 Temporal Redundancy

The response of the algorithm to temporal redundancy should be clear by now. The network processes data elements only when they change; it stores the information about the data elements as long as they remain unchanged.

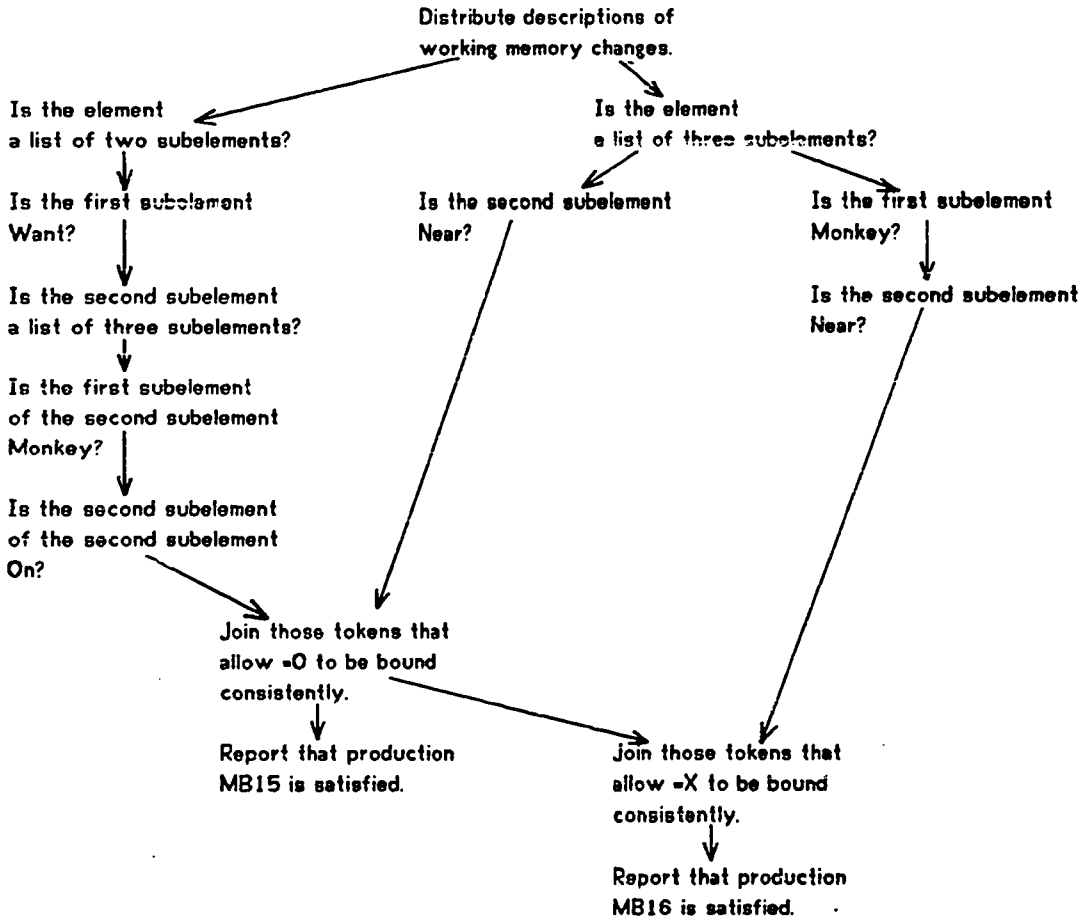
2.4.2 Structural Similarity

The Rete Match Algorithm takes advantage of structural similarity by sharing nodes between productions. When two LHSs compile into sequences of nodes containing identical initial subsequences, the initial subsequences are shared. The sharing is heaviest, of course, when productions are very similar. MB15 and MB16, for example, contain two identical condition elements.

```
MB15    ( (Want (Monkey On =0)) (=0 Near =X)
          -->
          (Want (Monkey Near =X)) )

MB16    ( (Want (Monkey On =0)) (=0 Near =X) (Monkey Near =X)
          -->
          (Want (EmptyHanded Monkey)) )
```

When these two productions are compiled together, most of the nodes in the network are shared.

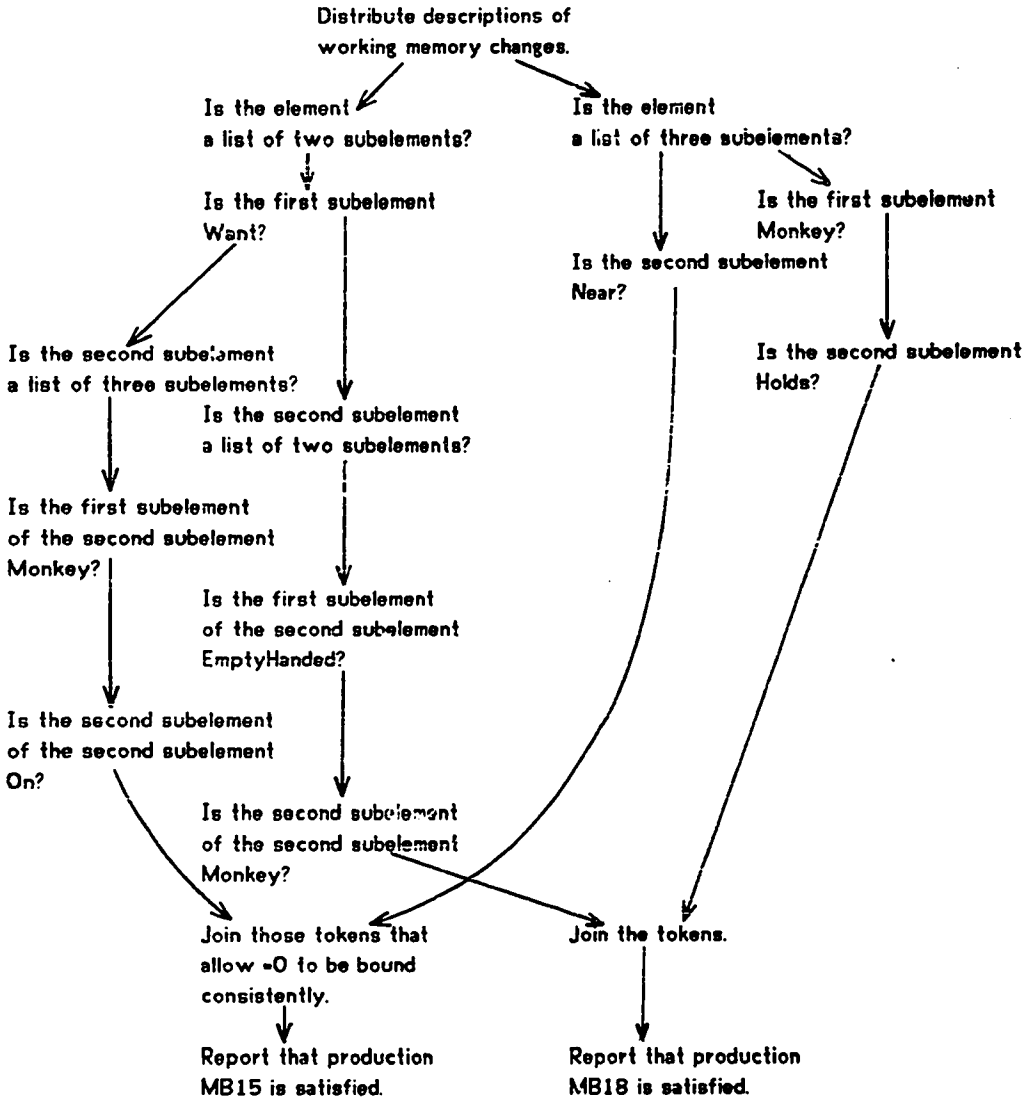


Even dissimilar productions often allow some sharing:

```
MB15    ( (Want (Monkey On =O)) (=O Near =X)
          -->
          (Want (Monkey Near =X)) )

MB18    ( (Want (EmptyHanded Monkey)) (Monkey Holds =X)
          -->
          (<WRITE> "The monkey drops the" =X)
          (<DELETE> (Want (EmptyHanded Monkey)) )
          (<DELETE> (Monkey Holds =X) ) )
```

The network for MB15 and MB18 contains seventeen nodes; it would contain twenty if nodes were not shared.



Although this chapter is not concerned with details like the representation of nodes in the interpreter, one comment about the representation should be made here: if variables are represented as they have been in the example networks of this chapter, some potential for sharing is lost. Consider productions MB15 and MB16 again. Suppose that MB15 was replaced by MB15', which is identical to MB15 except that the variable =O is renamed to =Z.

```
MB15'   ( (Want (Monkey On =Z)) (=Z Near =X)
          -->
          (Want (Monkey Near =X)) )

MB16    ( (Want (Monkey On =O)) (=O Near =X) (Monkey Near =X)
          -->
          (Want (EmptyHanded Monkey)) )
```

MB15' is functionally identical to MB15; it matches the same data elements, and it performs the same processing. In general, renaming a variable has no effect provided the renaming is carried out uniformly throughout the production and provided the variable is not given a name that already exists in the production. But with the representation of nodes assumed in this chapter, renaming a variable would have an effect on the network built by the LHS compiler. Since the variable binding nodes contain the names of the variables, renaming the variables could affect the sharing of these nodes. In the network for MB15' and MB16, for example, no two-input nodes could be shared. This effect of variable names on sharing can be eliminated, however. The names of the variables can be replaced in the nodes by integers that indicate where the variables occur in the condition elements.

2.4.3 Parallelism

This subsection discusses parallel execution of the algorithm. It considers two kinds of parallelism: processing many nodes at once and processing more than one working memory change at once. It shows that if the nodes are defined properly, then (1) no synchronization will be needed beyond that provided by the tokens, and (2) the results computed by the match will not depend on the order in which nodes with pending inputs are executed, or on the number of nodes active at one time.

The easiest way to examine the effects of parallel execution is to consider the different kinds of nodes in the order in which they occur in the networks. The first levels of the network contain the one-input nodes. The next levels contain the two-input nodes. The final level of the network contains the nodes for changing the conflict set.

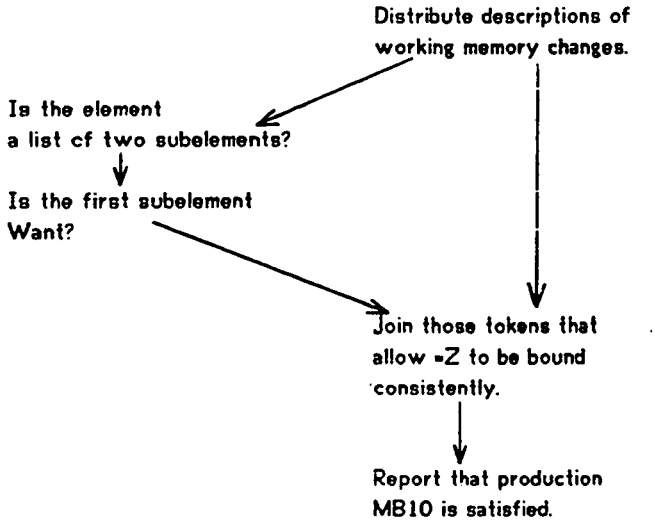
The one-input nodes certainly work properly when the match is executed in parallel. Since

a one-input node maintains no state, the processing it performs when activated by some token depends only on the specification of the node and the contents of the token. Other activity in the network and the history of the node have no effect.

The next nodes to consider are the two-input nodes that have only one-input nodes for predecessors. Since these nodes maintain state, they are sensitive to the order in which their inputs arrive. An example will show the effect of the order. Consider the network for MB10.

```
MB10    ( (Want =Z) =Z --> (<DELETE> (Want =Z)) )
```

The network is



At the beginning of a run of MKYBAN, the token

```
<VALID, (Want (Monkey Holds Bananas))>
```

is processed by the network. This token arrives at the left input of MB10's two-input node. Nothing that arrives on the right input will allow =Z to be bound consistently until the end of the run when MB4 fires. When this production fires it causes the network to process two tokens

```
<INVALID, (Want (Monkey Holds Bananas))>
<VALID, (Monkey Holds Bananas)>.
```

If

<VALID, (Monkey Holds Bananas)>

reaches the two-input node first, the node will send out the token

<VALID, (Want (Monkey Holds Bananas))(Monkey Holds Bananas)>

which will result in a new instantiation being added to the conflict set. Then when

<INVALID, (Want (Monkey Holds Bananas)>

arrives at the two-input node, the node will send out

<INVALID, (Want (Monkey Holds Bananas))(Monkey Holds Bananas)>

which will cause the instantiation to be removed from the conflict set. If the node had received the tokens in the other order, however, it would have produced no outputs. Thus in both cases, the final contents of the conflict set are unchanged, but in one case the node outputs a token and then immediately outputs another one with the opposite tag to reverse the effects of the first. These pairs of tokens with identical data parts and complementary tags are called conjugate pairs.

Before continuing with the two-input nodes, it should be noted that this example shows a limit on the ability of the algorithm to execute in parallel. Ideally, there would be no dependence on the order in which the pending tokens were chosen for processing. But in the example above, if

<INVALID, (Want (Monkey Holds Bananas))(Monkey Holds Bananas)>

arrived at the last node before

<VALID, (Want (Monkey Holds Bananas))(Monkey Holds Bananas)>

was processed, and then if the node processed the INVALID token before the VALID token, the conflict set would be changed incorrectly. When the INVALID token was processed, nothing could be removed from the conflict set. Then when the VALID token was processed, an instantiation of MB10 would be added. To prevent this problem, the edges must deliver the tokens to the nodes in the order they are placed on the edges.

To return now to the two-input nodes succeeding the one-input nodes: Since the assumption has been made that the edges maintain the order of the tokens, it is necessary only to consider tokens arriving on different inputs. A case analysis will show that the only

effect of order (other than the obvious effect of changing the order of the output tokens) is that conjugate pairs are sometimes produced. The case analysis will involve two tokens, L, which arrives on the node's left input, and R, which arrives on its right input. L and R allow consistent variable bindings. (Tokens that do not allow consistent variable bindings can be ignored since they do not interact with one another.)

The first case is that both tokens are tagged VALID. With this tag, L and R will both be stored into the node memories when they are processed. Observe that when a node quiesces after processing an input token, every element in the right memory has been tested against every element in the left, and every element in the left memory has been tested against every element in the right. Thus a test will be performed between L and R when the second one of the two arrives. Whether an output token is produced after the test does not depend on the order of their arrival if the node is one of the ordinary two-input nodes. If the node is a <NOT> node, and if R is the only element in the right memory that allows consistent variable bindings with L, then a conjugate pair is produced when L arrives first. When L arrives, it is output by the node; when R arrives, the conjugate of L is output.

The second case is that both L and R are tagged INVALID. These tokens could not arrive unless their data parts were already stored in the node memories. Since the first to arrive causes its data part to be removed from the memories, the test between their data parts occurs only once -- when the first token is processed. If this is an ordinary two-input node, whether tokens are output does not depend on the order of arrival. If this is a <NOT> node, and if R is the only element in the right memory that allows consistent variable bindings with L, then a conjugate pair is produced when R arrives first.

The third case is that L is tagged VALID and R is tagged INVALID. When L is processed, an element is stored in the left node memory. When R is processed, an element is deleted from the right node memory. If the node is an ordinary two-input node, a conjugate pair is produced when L arrives first. If the node is a <NOT> node, whether tokens are output does not depend on the order.

The final case is that L is tagged INVALID and R is tagged VALID. When L is processed, an element is deleted from the left node memory. When R is processed, an element is added to the right memory. If the node is the ordinary kind of two-input node, a conjugate pair is produced when R arrives first. If the node is a <NOT> node, whether tokens are output does not depend on the order.

The two-input nodes that are successors to other two-input nodes must cope with two effects of parallel processing: getting their inputs in an arbitrary order and sometimes receiving conjugate pairs. The effects of token arrival order are as they were for the other

two-input nodes -- conjugate pairs are sometimes produced. It can easily be seen that the effects of conjugate pairs cancel out. When the first token of a conjugate pair arrives at a two-input node, it causes a change in the node's internal state, and possibly causes some tokens to be sent out. Because the tags are treated symmetrically by the two-input nodes, when the second arrives, it will cause the node's state to change back, and if tokens were output before, it will cause the conjugates of these to be output.

The final kind of node to be considered is the node for changing the conflict set. Since the only actions performed by this node are adding elements to and deleting elements from a set, the order in which the tokens arrive at these nodes certainly does not matter. Neither, as the example above showed, does the processing of conjugate pairs.

In summary, the Rete Match Algorithm is capable of parallel execution provided the edges maintain the order of the tokens passed over them.

2.5 The Node Programs

The preceeding sections have raised a series of issues and described the Rete Match Algorithm's response to each issue. The result is a complete, but long description of the essential features of the algorithm. This section contains a more concise description.

The simple Rete Match Algorithm described in this chapter uses six classes of nodes. These are the bus node (the node that reports working memory changes to the rest of the nodes); the one-input nodes that test variable bindings; the one-input nodes that test constant features (e.g., the length of a list or the identity of an atom); the two-input nodes for non-negated condition elements; the <NOT> nodes; and the nodes that report changes to be made to the conflict set. Each of these classes is described below by listing two things: the information that is built permanently into a node of that type, and the processing performed a node when it is activated.

2.5.1 The One-input Nodes for Testing Constant Features

Built into a one-input node for testing constant features are

- The index of the subelement to test.
- A constant to compare the subelement (or its length) to.
- A description of the test to be performed.
- A list of the edges leaving the node.

When the node is activated by the arrival of a token, it performs three steps and then halts, passivating itself to await the next token:

1. Extract the subelement to be tested.
2. Perform the test.
3. If the test succeeded, send the token to the successors.
4. Halt.

2.5.2 The One-input Node for Testing Variable Bindings

Built into a one-input node for testing variable bindings are

- The indices of the two subelements to test.
- A description of the test it is to perform.
- A list of the edges leaving the node.

When the node is activated it performs the following steps:

1. Extract the first subelement.
2. Extract the second subelement.
3. Perform the test.
4. If the test succeeded, send the token to the successors.
5. Halt.

2.5.3 The Ordinary Two-input Node

The amount of information built into a two-input node depends on the number of variables tested by the node. For each variable to be tested, it has:

- Two indices of subelements.
- A description of the test to perform on the two subelements.

In addition, it has the usual links to its successors:

- A list of edges leaving the node.

The processing performed by the node depends on the edge over which the token arrived. If it arrived from the left:

```
1. If the token is tagged "VALID", store the data part in
the left memory; otherwise find and delete an identical
data part from the left memory.
2. Foreach data part in the right memory
        Begin
        3. Set FAIL = 0.
        4. Foreach variable to test until FAIL = 1
                Begin
                5. Use the first index to extract a subelement
                from the left data part.
                6. Use the second index to extract a subelement
                from the right data part.
                7. Perform the test.
                8. If the test failed, set FAIL = 1.
                End
        9. If FAIL = 0,
                Begin
                10. Set D = the data part of the token
                from the left concatenated with the data
                part of the token from the right.
                11. Build a token using D and the tag part
                of the token that just arrived.
                12. Send the new token to the successors.
                End
        End
13. Halt.
```

The processing performed when the token arrives from the right is similar, except that in some places (but not all -- see steps 5, 6, and 10) the roles of left and right are reversed.

```
1. If the token is tagged "VALID", store the data part in
the right memory; otherwise find and delete an identical
data part from the right memory.
2. Foreach data part in the left memory
        Begin
        3. Set FAIL = 0.
        4. Foreach variable to test until FAIL = 1
                Begin
                5. Use the first index to extract a subelement
                from the left data part.
                6. Use the second index to extract a subelement
                from the right data part.
                7. Perform the test.
                8. If the test failed, set FAIL = 1.
                End
        9. If FAIL = 0,
                Begin
                10. Set D = the data part of the token
                from the left concatenated with the data
                part of the token from the right.
```

```
                    11. Build a token using D and the tag part
                    of the token that just arrived.
                    12. Send the new token to the successors.
                    End
            End
13. Halt.
```

2.5.4 The <NOT> Node

A <NOT> node holds the same information as a regular two-input node. It has the usual links to its successors:

- A list of edges leaving the node.

For each variable to be tested, it has:

- Two indices of subelements.

- A description of the test to perform on these two.

When a <NOT> node receives a token on its left input it performs the following:

```
1. Set COUNT = 0.
2. Foreach data part in the right memory
        Begin
        3. Set FAIL = 0.
        4. Foreach variable to test until FAIL = 1
                Begin
                5. Use the first index to extract a subelement
                from the left data part.
                6. Use the second index to extract a subelement
                from the right data part.
                7. Perform the test.
                8. If the test failed, set FAIL = 1.
                End
        9. If FAIL = 0, set COUNT = COUNT + 1.
        End
10. If the token that just arrived is tagged "VALID", store
COUNT and the data part in the left memory; otherwise delete
an identical data part and its count.
11. If COUNT = 0, send the token that just arrived to the
successors.
12. Halt.
```

The processing performed when a token arrives from the right is quite different:

```
1. If the token is tagged "VALID", store the data part in
the right memory; otherwise find and delete an identical
data part from the right memory.
2. If the token is tagged "VALID", set INC = 1; otherwise
set INC = -1.
3. Foreach data part in the left memory
        Begin
        4. Set NEWCOUNT = the count stored with the data part.
        5. Set FAIL = 0.
        6. Foreach variable to test until FAIL = 1
                Begin
                7. Use the first index to extract a subelement
                from the left data part.
                8. Use the second index to extract a subelement
                from the right data part.
                9. Perform the test.
                10. If the test failed, set FAIL = 1.
                End
        11. If FAIL = 0,
                Begin
                12. Set NEWCOUNT = NEWCOUNT + INC.
                13. Replace the count in the left memory with
                NEWCOUNT.
                14. If (NEWCOUNT = 0 and INC = -1)
                or (NEWCOUNT = 1 and INC = 1),
                        Begin
                        15. If INC = -1, set TAG = "VALID";
                        otherwise set TAG = "INVALID".
                        16. Build a token using TAG and the
                        data part from the left.
                        17. Send the new token to the successors.
                        End
                End
        End
18. Halt.
```

2.5.5 The Node That Changes the Conflict Set

The node that changes the conflict set has only one item of information built into it

- The name of the production it represents.

The processing it performs upon the arrival of a token depends on the tag of the token:

```
1. If the token is tagged "VALID", add the data part of
the token and the production name to the conflict set;
otherwise, remove an identical entry from the conflict set.
2. Halt.
```

2.5.6 The Bus Node

The node that reports working memory changes to the rest of the network has built into it only one thing:

- A list of the edges leaving the node.

This node cannot be activated in the same manner as the other nodes. No tokens can be sent to it since this is the node where the first tokens are built. Exactly how it is activated depends on the structure of the rest of the monitor, so that will not be shown here. The assumption is made simply that it is activated after every working memory change. If the only changes made to working memory are adds of new elements and deletes of existing elements, the processing performed upon its activation is:

```
1. If the working memory change was an add, set TAG = "VALID";
otherwise, set TAG = "INVALID".
2. Build a token using the affected working memory element
and TAG.
3. Send the token to the successors.
4. Halt.
```

2.6 The Range of Applicability of the Algorithm

Although this description of the Rete Match Algorithm has been motivated solely by examples from OPS2, the algorithm can be used for other languages as well, including languages quite unlike OPS2. The following paragraphs describe the range of applicability of the algorithm; they consider individually (1) the kinds of data elements that can be processed, (2) the kinds of subelement tests that can be used (that is, what can be used to compose a condition element), and (3) the allowable ways of combining condition elements to construct LHSs.

There is a fundamental restriction on the kinds of data elements that can be processed by the algorithm, but the restriction is not that they must be OPS2-style lists. The restriction is that they must contain only constants. Variables and match functions cannot be handled without revising the nodes and the network interpreter. That the algorithm is not restricted to list structured data elements can be argued most easily by appealing to the generality of OPS2. Appendix II shows that OPS2 data elements can simulate name-attribute-value triples, strings, sets, and other data types. Since the transformations between formats are purely mechanical, it would be possible to write a program that would accept productions and data

elements in one of the other formats and translate them into OPS2 productions and data elements. This translator together with the OPS2 interpreter would comprise an interpreter for production systems that use the other data format.

The subelement tests are limited in one important respect: the tests must be independent. The tests cannot cooperate because the nodes are not allowed to communicate except by sending tokens. This limitation has two particularly important manifestations. First, partial matches cannot be computed. For example, the algorithm as defined here would not support a language that allowed matches between elements provided a certain percentage of their constant subelements were equal. Second, it must be possible to determine at compile time which data subelement each condition subelement will match. For example, if the symbol "..." were defined to match any arbitrary segment from a list, the following production could not be compiled.

```
Unc    ( (... A ... =X ... =X ... A ...) --> =X )
```

This requirement of independence is the only limit to the subelement tests. The Rete Match Algorithm can handle any test which will accept a data subelement (two subelements in the case of the variables) and return TRUE or FALSE. In fact, the OPS2 interpreter allows the user to define his own tests, which are compiled into the network as the predefined tests are.

Condition elements can be combined in almost any fashion when the Rete Match Algorithm is used. OPS2, for example, allows almost arbitrary use of conjunction and negation. An OPS2 LHS may contain arbitrarily many condition elements. Negation may apply both to single condition elements and to conjunctions of other condition elements, including conjunctions that contain other instances of negation. And conjunctions and negations may be nested to arbitrary depths. Adding disjunctions of condition elements would entail a few changes to the compiler, but the network interpreter could be used unchanged.

3. The OPS2 Interpreter

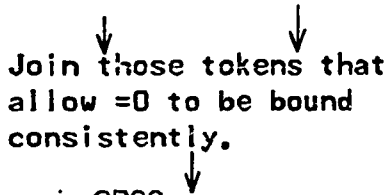
This chapter contains a detailed description of the OPS2 interpreter, the most recent production system interpreter to use the Rete Match Algorithm.[1] The information contained in this chapter is necessary to understand the cost analyses made in the next chapter. Chapter 3 begins by explaining two major differences between the algorithm described in Chapter 2 and the OPS2 match algorithm. Then it describes the format of the nodes and tokens in OPS2. Finally it describes the processing performed by the nodes. The description in this chapter is complete enough to allow the reader to duplicate the match part of the OPS2 interpreter.

3.1 Reducing the Cost of Storing Tokens

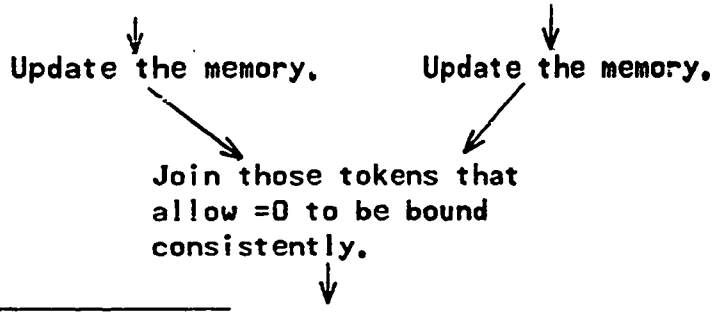
Because the OPS2 interpreter was designed to run on a uniprocessor, the algorithm from Chapter 2 could be changed in a way which significantly reduces the number of tokens stored in the network. This section describes the change.

3.1.1 Separate Memory Nodes

The two-input nodes described in the last chapter perform two related but distinct functions. They maintain internal memories containing the data parts of tokens, and they compare pairs of data parts to determine which allow consistent variable bindings. Although most interpreters using the Rete Match Algorithm do have two-input nodes of this kind, some versions use simpler nodes. In the OPS2 interpreter, three simple nodes are used to perform the functions of one of the more complex two-input nodes. One node maintains the left memory, a second maintains the right memory, and a third performs the variable binding tests. A typical node from the last chapter



becomes in OPS2:



[1] The OPS2 language is described in [22].

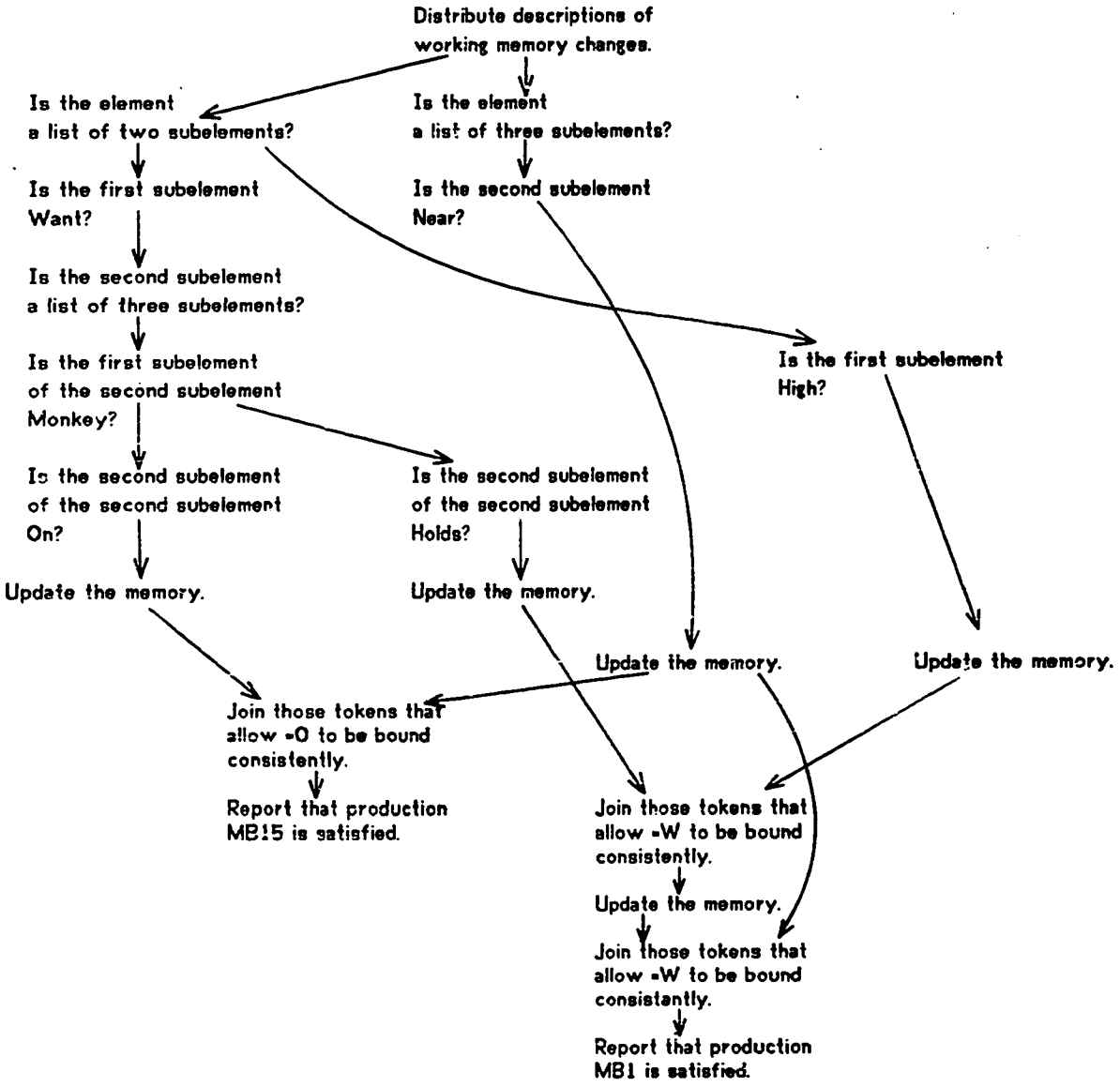
Instead of two internal memories, a two-input node in OPS2 contains pointers back to its two predecessors. The predecessors are memory nodes.

Memories have been separated out of the two-input nodes because this increases the potential for sharing. Memory nodes can be shared between two productions if the productions have condition elements that are identical except for variables. The third condition element in MB1 and the second condition element in MB15, for example, allow sharing a memory node.

```
MB1     ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
          -->
          (Want (Ladder Near =P)) )

MB15    ( (Want (Monkey On =O)) (=O Near =X)
          -->
          (Want (Monkey Near =X)) )
```

The network for these two productions contains five memories; six would have been required if the memories were internal to the two-input nodes.



Breaking up the two-input nodes improves the efficiency of the algorithm in some ways, but degrades the efficiency in one. Both the time required to update the network on each cycle and the number of tokens stored in the network are reduced by breaking up the nodes. But even with sharing of the memory nodes, the total number of nodes in the network is increased.

3.1.2 Memories for <NOT> Nodes

The <NOT> nodes do not share their left memories. Recall that the left memory of a <NOT> node stores a count along with each data part. Since these counts have meaning only to the node that generated them, they would present problems if stored in a shared structure. Thus

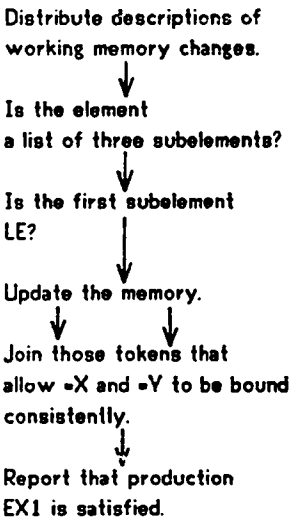
the <NOT> nodes should store the counts internally. The <NOT> nodes could store only the counts, relying on memory nodes to store the data parts. But when the OPS2 interpreter was designed, it was deemed simpler to have the data parts and the counts stored together. Hence the left memory of the <NOT> node was kept internal to the node. Since the right memory of a <NOT> node stores only data parts of tokens, it can be shared like the memories of a regular two-input node.

3.1.3 The Processing Performed by the Memory Nodes

The memory nodes perform only three functions. They maintain their internal memories, they pass tokens along edges leading to the right inputs of two-input nodes, and they pass tokens along edges leading to the left inputs of two-input nodes. The order in which these operations are performed is critical. Consider the following production:

```
EX1     ( (LE =X =Y) (LE =Y =X) --> (EQ =X =Y) ).
```

Because the two condition elements are identical except for the order of the variables, the two-input node in EX1's network shares a memory node with itself.



If the element **(LE P P)** entered an empty working memory, EX1 would have a legal instantiation, and this network should find it. But if the memory node performs its operations in the wrong order, the network might either fail to add the instantiation to the conflict set or add it twice.

Suppose the memory node modified its internal memory before sending the token to its successors. Then when the token

<VALID, (LE P P)>

arrived at the memory node, the node would store away the data part of the token and then send the token over every edge leaving the node. Since two edges connect the memory node to the two-input node, the token would arrive at the two-input node twice. If it arrived on the left first, the node would look at its right memory and find **(LE P P)** there. Since the variables can be bound consistently, the node would send out the token

<VALID, (LE P P)(LE P P)>.

When the token

<VALID, (LE P P)>

arrived on the right input, the node would check its left memory, find **(LE P P)** there, and again send out the token

<VALID, (LE P P)(LE P P)>.

The last node would thus receive two identical tokens and add two instantiations of EX1 to the conflict set.

Since the match proceeds incorrectly if the memory nodes modify their internal memories before sending out the tokens, suppose that the nodes modify their memories after sending the tokens. If this is to be deterministic, it must be assumed that the memory nodes allow their successors to finish processing before modifying their memories. Then when the memory node received the token

<VALID, (LE P P)>

it would immediately send it out along the edges leaving the node. If the token arrived first at the left input of the two-input node, the node would check its right memory, find nothing there, and produce no outputs. When the token arrived at the right input, the node would check its left memory, find that empty also, and again produce no outputs. Finally the memory node would modify its memory, too late for the two-input node to produce an output.

The match will proceed correctly only if the memory node distinguishes the edges that reach right inputs of nodes from the edges that reach left inputs of nodes. One correct sequence is first send the token out along the edges leading to left inputs of two-input nodes, wait for all the nodes to finish processing, then modify the node memory, and finally

send the token out along the edges leading to the right inputs. If the memory node did this, when

<VALID, (LE P P)>

arrived, it would first send the token along the edge leading to the left input of the two-input node. The two-input node would check its right memory and, finding nothing there, produce no output. When the two-input node finished processing, the memory node would add **(LE P P)** to the memory and send the token out along the edge leading to the right input of the two-input node. The two-input node would check its left memory and find **(LE P P)** there. Since the variables can be bound consistently, it would output the token

<VALID, (LE P P)(LE P P)>.

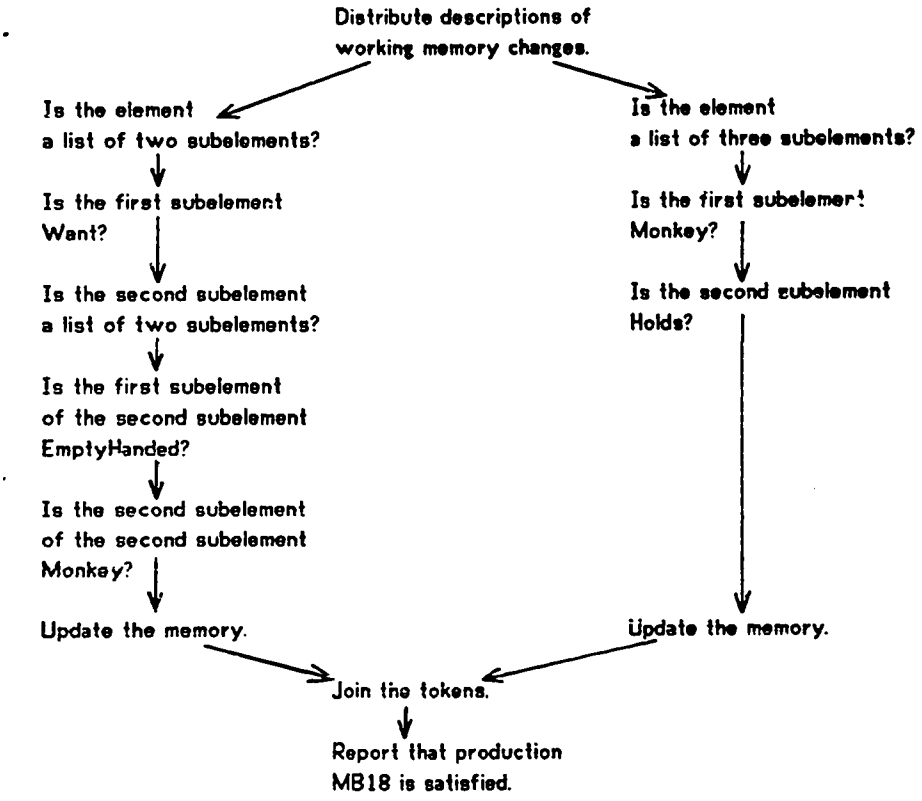
Since the last node in the network would receive only this one token, it would add the instantiation of EX1 to the conflict set only once.

3.1.4 Synchronizing the Divided Two-input Nodes

The separate memory nodes were not described in the last chapter because their use makes the algorithm sensitive to the order in which nodes are executed. This sensitivity presents little problem for a uniprocessor implementation like OPS2, but it would complicate the design of a parallel interpreter. To see an example of the problem, consider production MB18 again.

```
MB18    ( (Want (EmptyHanded Monkey)) (Monkey Holds =X)
          -->
          (<WRITE> "The monkey drops the" =X)
          (<DELETE> (Want (EmptyHanded Monkey)) )
          (<DELETE> (Monkey Holds =X) ) )
```

The OPS2 compiler would translate MB18's LHS into:



Suppose **(Want (EmptyHanded Monkey))** and **(Monkey Holds Ladder)** entered an empty working memory at the same time, and suppose the match processed both these changes in parallel. If both the tokens reach the two-input node simultaneously, the order in which the tokens are processed will determine what is output by the node. If the two-input node accepts the token from the left first, it may produce two output tokens. When it processes

<VALID, (Want (EmptyHanded Monkey))>

it will check its right memory and find the data part of

<VALID, (Monkey Holds Ladder)>.

The data part will be there because, if the memory node used the order of processing decided upon in the last section, it would have modified its memory before sending the token to the right input of a node. Since it finds the data part in the right memory, the two-input node will build and output the token

<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Ladder)>.

After the two-input node halts, two things happen in parallel. The two-input node will accept the other token, and the memory node on the left branch of the network will modify its memory. If the left memory is modified before the two-input node reads it, the two-input node will find there the data part of

<VALID, (Want (EmptyHanded Monkey))>.

This will cause the two-input node to again output the token

<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Ladder)>.

Since there OPS2 interpreter executes serially, it is not difficult to order the processing of nodes in such a way that this problem is avoided. (See section 3.4.1.) If parallel execution were used, however, explicit synchronization of the nodes would be necessary.

3.2 A Third Action Type

A second difference between the algorithm in the last chapter and the OPS2 algorithm is that the OPS2 algorithm supports three action types rather than two. In addition to the add (which adds one element to working memory) and the delete (which removes one element from working memory), the OPS2 interpreter supports an action called reassert. Like the delete action, the reassert is performed only on elements that are present in working memory at the time the RHS is executed. Reasserting an existing element is defined to be equivalent to deleting the element and then adding an identical element. The monitor could convert the reassert into the two actions internally, and then the match could process them using the methods described in the last chapter. But if the nodes are given the ability to handle more token types, the reassert can be processed directly. This allows processing one action to give the effect of processing two.

3.2.1 A Third Tag

The processing of the third action is consistent in form with the processing of the other two actions. The processing begins with the creation of a token whose data part holds the reasserted element and whose tag indicates that the action was a reassert. This tag is OLD-VALID. The token is sent out along the edges leaving the bus node of the network, and the nodes it reaches are activated to perform some processing. This section describes the processing performed by the nodes.

First the overall effect must be considered. Since a reassert is defined to be equivalent to

a delete followed by an add, the contents of working memory are unchanged. If working memory is unchanged, the node memories should be also. The conflict set is changed only in that the effects of previous applications of conflict resolution rules must be modified. Instantiations containing the reasserted element must be reevaluated under the recency criteria since the recency of the element has changed. If any instantiations containing the element are marked as having been fired (and thus ineligible to fire again under the OPS2 conflict resolution rules) the mark must be removed. The question to be answered in this section, then, is how the nodes in the network can process a reassert and change nothing but the conflict set.

The one-input nodes should treat tokens with the new tag exactly like the other tokens. Since they do not consider the other two tags, they should not consider this one.

Since the memories in the network are supposed to remain unchanged when a reassert is processed, the memory nodes should do nothing more than pass the OLD-VALID tokens on to their successors.

The two-input nodes excluding the <NOT> nodes must treat OLD-VALID tokens like the other tokens. They must build new data parts as they did in processing the other tokens, because the routines that maintain the conflict set must receive the complete data parts of the instantiations if they are to know what to change. If a two-input node builds a token to [illegible] during the processing of an OLD-VALID token, it must be the case that another token wi[illegible]dentical data part was produced earlier (when the last of the data elements involved en[illegible] working memory). Thus the tokens built while processing a reassert must be given the tag that will leave the memories below unchanged: OLD-VALID.

When a <NOT> node receives a token tagged OLD-VALID on its left input, it must not change its internal memory because all memories in the network, whether they are contained in <NOT> nodes or memory nodes, must remain unchanged. In deciding whether to pass the token on to its successors, it must ignore the tag just as the one-input nodes do. It must compare the data part of the token to the elements stored in its right memory to determine how many allow consistent variable bindings, and if the number is zero, it must send the token (unchanged) to its successors.

A <NOT> node cannot produce an output while processing an OLD-VALID token that arrives from the right. If the token has that tag, the data part of the token is necessarily already stored in the right memory. Hence every element in the left memory that allows consistent variable bindings has a count of at least one. Since processing the reassert cannot lower the counts, no tokens can be output.

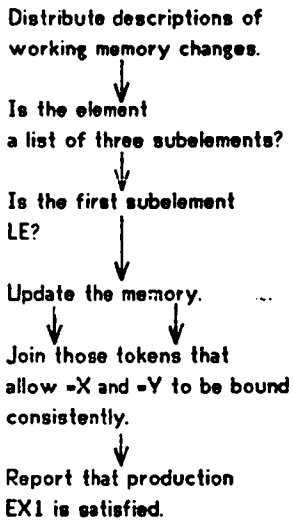
Some details of the OPS2 conflict set must be explained before the processing of OLD-VALID tokens can be described. The conflict set in OPS2 is an ordered list of the instantiations that have not yet been executed. Instantiations are removed from the set when they are executed because the OPS2 conflict resolution rules specify that an instantiation may not execute twice. The order of the list is the order of dominance under the other conflict resolution rules; each instantiation dominates all those later in the list. The routines that maintain the conflict set preserve the order of the elements when making changes to the set. When they receive an instantiation that resulted from the processing of a reassert, the routines must perform two operations to keep the set in correct order. First, they must remove any identical instantiations because, with the recency of one of the data elements just changed, the old position of the instantiation is probably no longer correct. (Of course, if the instantiation had executed, it would no longer be in the set.) Then they must place the new instantiation into the conflict set at the appropriate position.

3.2.2 Inverting Tags at <NOT> Nodes

The <NOT> nodes described in the last chapter sometimes had to invert tags. The arrival on the right of a VALID token resulted in sending out INVALID tokens; the arrival on the right of an INVALID token resulted in sending out VALID tokens. The <NOT> nodes in OPS2 must invert tags also. It might appear that the existence of a third tag would complicate the process of inversion, but the third tag actually has no effect on the process. Recall that the arrival on the right input of a token tagged OLD-VALID cannot cause any outputs to be produced. Hence there is no need to invert this tag. Neither of the other tags ever becomes OLD-VALID because the arguments regarding changing VALID tags to INVALID and INVALID tags to VALID still apply. If a token tagged VALID arrives on the right and causes the count of some element in the left memory to change from zero to one, then the token produced should be tagged INVALID. Since the element had a count of zero, it must have been stored in the memories below this node at some earlier time. The INVALID token is needed to cause the memory nodes to remove the element. Similarly, if a token tagged INVALID arrives and causes a count to change from one to zero, the token produced should be tagged VALID. If the element had a count of one, it could not be in the memories below this node. The VALID tag is needed to cause the memory nodes to store the element.

3.2.3 OLD-VALID Tokens and Shared Memories

The two changes made to the basic algorithm in developing the OPS2 algorithm interact to produce an unfortunate effect: the nodes in the network sometimes produce more output tokens than they should. To see the effect, consider the network for EX1 again.



Suppose that the data element **(LE P P)** was reasserted. When the token

<OLD-VALID, (LE P P)>

reached the left input of the two-input node, the node would examine its right memory, find **(LE P P)** there (because the data element was added to working memory on an earlier cycle), and produce the following output token.

<OLD-VALID, (LE P P)(LE P P)>

When

<OLD-VALID, (LE P P)>

reached the right input of the two-input node, the node would examine its left memory and again output the two-element token.

<OLD-VALID, (LE P P)(LE P P)>

This multiplying of tokens could be eliminated if the memory nodes deleted the data part from their memories before sending the token to the left inputs of the following nodes and then replaced it before sending the token to the right inputs of the following nodes. But the decision was made that the memory nodes should not engage in this extra activity. Making these extra changes to the node memories would slow the interpreter, and the extra tokens can be dealt with more easily at the node that changes the conflict set. (Since the node memories are not affected by the processing of OLD-VALID tokens, the only potential effect

of the extra tokens would be changes in the conflict set.)

3.3 The Interpreter's Data Formats

This section describes the form of the two kinds of data processed by the OPS2 interpreter: tokens and nodes.

3.3.1 The Token Format

Tokens in OPS2 are lists. The first element of the list is the tag of the token, and the remaining elements are pointers to data elements. The token

```
<VALID, (Want (EmptyHanded Monkey))>
```

is represented in OPS2

```
(VALID (Want (EmptyHanded Monkey))).
```

The data parts of the tokens are in reverse order. The token

```
<VALID, (Want (EmptyHanded Monkey))(Monkey Holds Ladder)>
```

is represented

```
(VALID (Monkey Holds Ladder)(Want (EmptyHanded Monkey))).
```

The data parts of the tokens are built backwards because this minimizes the amount of space consumed by the tokens. Consider the following data part.

```
((Bananas Near (8 2))(High Bananas)(Want (Monkey Holds Bananas)))
```

As stored in Lisp[1] this data part extends over fifteen words of memory. To store the top level list (the data part itself) requires three words. To store the first data element requires five words; to store the second, two words; and to store the third, five words. The space cost of storing this token, however, is much less; even if the token were not stored, fourteen of the fifteen words would still be needed for other purposes. The data elements must be stored in working memory. (The data part holds not copies of the elements, but pointers

[1] See the manual for Lisp 1.6 [41], the version of Lisp in which OPS2 is implemented.

back to the appropriate slots in working memory.) Two of the remaining three words are used in other tokens. Since the OPS2 compiler arranges the two-input nodes in a linear sequence, joined by their left inputs, the data part

```
((Bananas Near (8 2))(High Bananas)(Want (Monkey Holds Bananas)))
```

could not be in a node memory unless the data part

```
((High Bananas)(Want (Monkey Holds Bananas)))
```

were in the left memory of the node that built the longer token. This two-element data part, in turn, could not be in that memory unless

```
((Monkey Holds Bananas))
```

were stored in an earlier memory. Since Lisp allows lists to share common tails, the OPS2 interpreter can use the entire two-element data part as the tail of the three-element data part; and it can use the one-element data part as the tail of the two-element data part. As a result of the sharing, the three data parts consume a total of only three words of storage. Since this sharing of tails is used throughout the network, every token stored consumes only one word, regardless of its length.

3.3.2 The Node Formats -- Preliminaries

A node in OPS2 is a list of from two to six elements. The positions in the list are called fields. The contents of the first field indicate the function to be performed -- whether the node is to test the length of subelements, to test for the occurrence of a constant, to test variable bindings, or to perform one of about six other functions. The other fields hold pointers to other nodes, constants, and the other information needed by the node program to perform its function.

The OPS2 interpreter has gone through many revisions, and the contents of these fields have changed on occasion. In general, this chapter describes the version of OPS2 that is measured in Chapter 5. One exception is the field that indicates which subelement to test. In that version of the interpreter, the field holds a pointer into a table. Since this pointer would carry little information here, the fields are shown holding a simplified version of what was used in an earlier OPS2 interpreter: lists called index vectors. The first element of an index vector contained information that was relevant only to the variable-binding nodes. The rest of the list contained integers that pointed to subelements of the tokens' data parts. Encoded in the list of integers was an indication of whether to test the subelement or the sublist

beginning with that subelement. These examples will suppress some of the details. It will always be assumed that the subelement and not the list beginning with the subelement is tested, and the encoding of the distinction will not be shown. The encoding of the variable information will be shown only for the variable-binding nodes. At the nodes that do not test variables, the list **(1)** means test the entirety of the data element. The index is 1 to indicate the first data element of the data part -- indices greater than 1, of course, are needed only after two-input nodes build longer data parts. The list **(1 3)** means test the third subelement of the data element. The list **(1 3 2)** means test the second subelement of the third subelement of the data element.

The version of OPS2 measured in Chapter 5 does not differ greatly in how it handles index vectors. The index vectors are stored in a table, and the nodes hold pointers into this table. This allows the nodes to share index vectors.

3.3.3 The One-input Nodes

Four of the one-input nodes contain only four fields. These are the &LEN, &LEN+, &ATOM, and &VIN nodes, which are described in this subsection.

The &LEN node tests the lengths of data elements and subelements. The first field in one of these nodes contains the type of the node (&LEN). The second field contains the list of successors of the node. The third field contains an index vector; and the fourth contains an integer constant. The node

```
Is the second subelement
a list of three subelements?
```

is encoded in OPS2

```
(&LEN (. . .) (1 2) 3).
```

The ellipsis represents the pointers to the successors of the node.

The &LEN+ node is like the &LEN node except that it allows subelements whose lengths are equal to or greater than the integer constant. It performs the length test for condition elements that contain the OPS2 segment character (!). The node

```
Is the element
a list of one or more
subelements?
```

which would be needed for the condition element (Want ↓ =) is encoded in OPS2

```
(&LEN+ (. . .) (1) 1).
```

The &ATOM node tests for the occurrence of a particular constant. The node

```
Is the first subelement
Want?
```

is encoded

```
(&ATOM (. . .) (1 1) Want).
```

The &VIN node tests variables that occur more than once in a condition element. Its name was chosen to indicate that it tests variables that are internal to a condition element. The node

```
Is the first subelement
equal to
the third subelement?
```

is encoded

```
(&VIN (. . .) (VARIABLE 1 1) (VARIABLE 1 3)).
```

This node shows the encoding of variables when both are ordinary variables ("=" variables). The condition element (#X =X) compiles into the &VIN node

```
(&VIN (. . .) (NOTVARIABLE 1 1) (VARIABLE 1 2)).
```

3.3.4 User Defined Match Predicates

Instances of user defined match predicates compile into one-input nodes containing five fields. The first field holds the type of the node, &USER. The second field holds the list of successors of the node. The third field holds the index vector. The fourth field holds the name of the user defined predicate, and the fifth field holds the list of arguments to the predicate. Production MB14, which contains the user defined predicate <NOTANY>, is an example of a production that needs the &USER node.

```
MB14    ( (Want (Monkey On Floor))
          (Monkey On =0 & (<NOTANY> Floor))
          -->
          (<WRITE> "The monkey jumps off of the" =0)
          (<DELETE> (Want (Monkey On Floor)) )
          (<DELETE> (Monkey On =0))
          (Monkey On Floor) )
```

The &USER node or this production is

```
(&USER (. . .) (1 3) <NOTANY> (Floor)).
```

3.3.5 The Memory Node

The memory node contains four fields. The first field, as always, holds the type of the node, &MEM. The second and third fields both hold lists of successors of the node. Recall that a memory node must distinguish the successors for which it plays the role of left memory from the ones for which it plays the role of right memory. It does so by having one field for what could be called the "left successors" and one for the "right successors." The last field holds a list of one element. That element is the memory of the node. Since OPS2 memories are simply lists of data parts, a memory node with an empty memory is

```
(&MEM (. . .) (. . .) ((())).
```

3.3.6 The &TWO Node

Although the left memory of a <NOT> node is internal, the left input of the node cannot come directly from the preceding one-input or two-input node. All the two-input nodes require that their immediate predecessors distinguish between left and right successors. Since the one- and two-input nodes do not make this distinction, they cannot be immediate predecessors of <NOT> nodes. Memory nodes could precede the <NOT> nodes, but it would be wasteful to put in a memory node just for this purpose. In OPS2 a node type has been defined that has no purpose but to distinguish its left and right successors. These nodes have three fields, one for the type of the node (&TWO) and two for the successors of the node. They are like &MEM nodes without memories:

```
(&TWO (. . .) (. . .)).
```

3.3.7 The Ordinary Two-input Node

The two-input nodes excluding the <NOT> nodes have six fields. The first field holds the type of the node, &VEX. (This name was chosen to indicate that the node tests variables which are externally bound.) The second field holds the list of successors of the node. The third and fourth fields hold pointers to the two predecessors of the node. These pointers are needed because the predecessors contain the memories read by the &VEX node. The fifth and sixth fields are lists of index vectors. Each variable checked by the node has one index vector in each list. The vectors in the fifth field point to subelements in the data from the left. The vectors in the sixth field point to a subelements in the data from the right. The production MB15

```
MB15    ( (Want (Monkey On =0)) (=0 Near =X)
          -->
          (Want (Monkey Near =X)) )
```

has one variable common to both condition elements. Its &VEX node therefore has only one test to perform.

```
(&VEX (. . .)(. . .)(. . .)((VARIABLE 1 2 3))((VARIABLE 1 1)))
```

3.3.8 The &NOT Node

The <NOT> nodes contain six fields, most of which have the same purpose as the corresponding fields in the other two-input nodes. The first field holds the type of the node, &NOT. The second field holds the list of successors of the node. The third field holds a pointer to the right memory of the node. The fourth field holds the left memory of the node (not a pointer to the memory). This field is identical to the fourth field of the &MEM node. The fifth and sixth fields hold lists of index vectors like the lists in the &VEX node. The production MB19

```
MB19    ( (Want (EmptyHanded =X)) - (=X Holds =)
          -->
          (<DELETE> (Want (EmptyHanded =X)) ) )
```

has one &NOT node in its network.

```
(&NOT (. . .) (. . .) (()) ((VARIABLE 1 2 2)) (VARIABLE 1 1)))
```

3.3.9 The &P Node

The nodes that report changes to the conflict set are the &P nodes. (This name was chosen to indicate that the node is the representative of the production.) These nodes have five fields. The first field holds the type of the node. The next three fields hold integers that are used in conflict resolution: a count of the number of condition elements in the LHS, a count of the number of constants in the LHS, and the ordinal number of the production (the ordinal number is one greater than the number of productions compiled prior to this one). The fifth field holds the name of the production. If MB1

```
MB1     ( (Want (Monkey Holds =W)) (High =W) (=W Near =P)
          -->
          (Want (Ladder Near =P)) )
```

is the first production to be compiled, its &P node is

```
(&P 3 5 1 MB1).
```

3.3.10 The &BUS Node

The first node in the network (the root of the entire network) has only two fields. The first holds the type of the node, &BUS. The second holds the list of successors of the node.

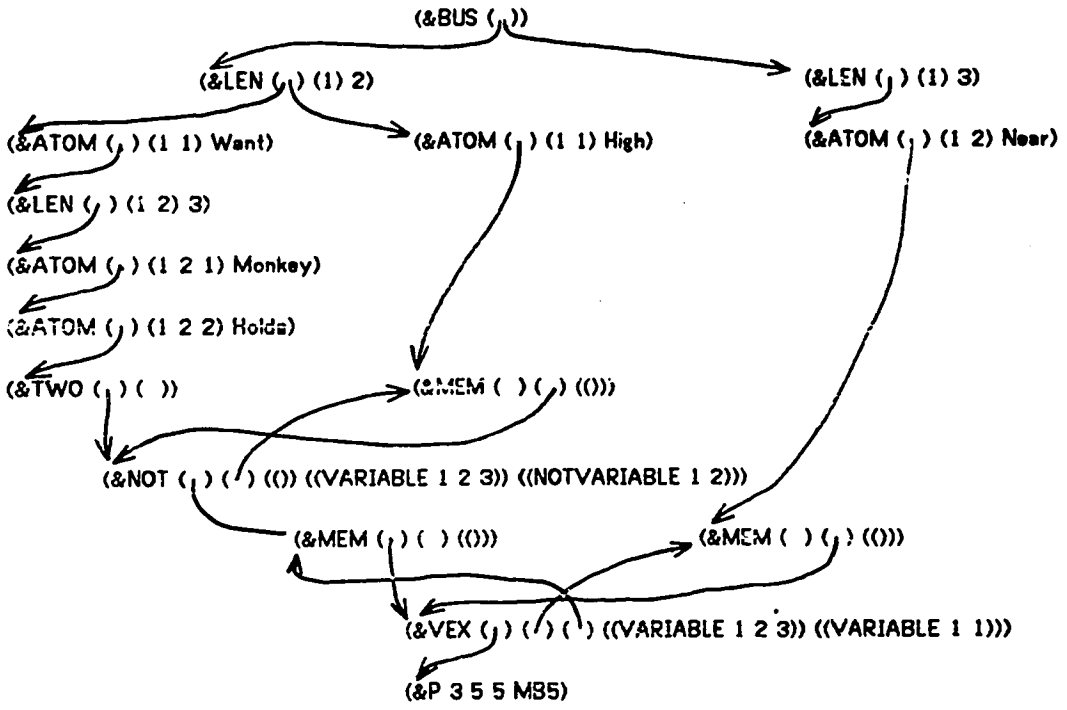
```
(&BUS (. . .))
```

3.3.11 An Example Network

The network for MB5 contains examples of every node type except &VIN, &LEN+, and &USER.

```
MB5     ( (Want (Monkey Holds =W)) - (High =W) (=W Near =P)
          -->
          (Want (Monkey Near =P)) )
```

When the node memories are empty, the network is



3.4 The Interpreter for the Nodes

The nodes in an OPS2 network can be considered instructions for an OPS2 machine. Just as a conventional computer interprets instructions like

```
MOVI R5, 0
ADD R1, R2
```

the OPS2 machine interprets instructions like

```
(&P 2 5 18 MB18)
(&VIN (. . .) (VARIABLE 1 3) (NOTVARIABLE 1 4)).
```

Although the instructions for the conventional computer are stored as bit vectors and the nodes for the OPS2 machine are stored as lists, they contain similar information. The name in the node's type field corresponds to the contents of the op-code field in a conventional instruction. The successor field is similar in function to the next-instruction field in the old four-address computers. The other fields are used like the data fields of immediate instructions.

The present OPS2 machine comprises an interpreter written in Lisp and a KL version of a PDP-10 computer on which the OPS2 interpreter is run. It functions much like a microprogrammed interpreter for a conventional architecture, with the PDP-10 playing the

role of the microprogrammable hardware, and the Lisp program playing the role of the microcode. The OPS2 interpreter fetches a node as the microprogrammed interpreter fetches an instruction. It examines the name field and branches to the appropriate subroutine as the other decodes the op-code field and branches to the appropriate microsubroutine. The following sections explain how this interpreter functions; they explain how it chooses which node to process when several are pending, they describe the subroutines for the various node types, and they explain how data is passed to the subroutines.

3.4.1 Control in the OPS2 Match

Since the OPS2 match is implemented on a uniprocessor, there is always only one active locus of control in the match. The interpreter follows three rules in choosing how to move the locus. First, it processes only one working memory change at a time. If three actions are performed on a cycle, the match is performed three times. Second, the interpreter selects the nodes to be executed in a depth-first manner. After executing one node, it executes all that node's descendants before executing any other nodes. Third, when a node sends out a token before it has finished processing, the interpreter suspends that node until all its successors have finished processing the token. When a memory node sends a token to its left successors, for example, it must wait until they finish processing before it can modify its memory and send the token to its right successors. As a result of the third rule, if a node produces more than one output token, each token is processed by all the node's successors before the processing of the next begins -- before, in fact, the next can be built.

These rules are easy to implement in a language supporting recursion, but that is not the reason they were chosen. They were chosen to avoid the potential problems of shared memories (see section 3.1.4). If these rules are followed, changes are made to memories one at a time, and every node below a memory is allowed to process the change before that memory -- or any other -- can be further modified. These rules, combined with the distinction between left and right successors made by the memory nodes, are sufficient to prevent the problems.

3.4.2 The One-input Nodes

The processing performed by the one-input nodes in an OPS2 network is similar to that performed by the one-input nodes in Chapter 2. The program shown here is different only because the information contained in the OPS2 nodes has been described in more detail.

Since the programs for &LEN, &LEN+, and &ATOM are identical except for the tests performed, only the program for &LEN is shown here.

1. Using the index vector, extract the subelement to be tested from the data part of the token.
2. Test whether the length of the subelement is equal to the constant parameter of the node.
3. If the test succeeded, send the token to the successors.
4. Halt.

3.4.3 The &VIN Node

The node program for &VIN differs from the corresponding program in Chapter 2 principally in that the &VIN program shows how the node choses which test to apply. Since OPS2 has four variable types, the node chooses from four possibilities, testing for equality, testing for inequality, testing for the first's being less than the second, and testing for the first's being greater than the second.

1. Using the first index vector, extract the first subelement from the data part of the token.
2. Using the second index vector, extract the second subelement from the data part of the token.
3. Using the variable-type part of both index vectors, choose the test to perform.
4. Perform the test.
5. If the test succeeded send the token to the successors.
6. Halt.

3.4.4 The &MEM Node

The &MEM node does not occur in the networks described in Chapter 2.

1. Send the token to the left successors.
2. If the token is tagged "VALID", store the data part in the node memory; otherwise if it is tagged "INVALID", find and delete an identical data part.
3. Send the token to the right successors.
4. Halt.

3.4.5 The &TWO Node

The &TWO node is also new to the OPS2 algorithm.

1. Send the token to the left successors.
2. Send the token to the right successors.

```
3. Halt.
```

3.4.6 The &VEX Node

The ordinary two-input nodes differ from the ones in Chapter 2 only in not containing memories. No change in the node program was needed to accommodate the third tag.

When an &VEX node receives a token from the left, it performs the following processing.

```
1. Foreach data part in the right memory
        Begin
        2. Set FAIL = 0.
        3. Foreach variable to test until FAIL = 1
                Begin
                4. Use the first index vector to extract a
                subelement from the left data part.
                5. Use the second index vector to extract a
                subelement from the right data part.
                6. Using the variable type part of both
                index vectors, choose which test to perform.
                7. Perform the test.
                8. If the test failed, set FAIL = 1.
                End
        9. If FAIL = 0,
                Begin
                10. Set D = the data part of the token
                from the left concatenated with the data
                part of the token from the right.
                11. Build a token using D and the tag part
                of the token that just arrived.
                12. Send the new token to the successors.
                End
        End
13. Halt.
```

When the node receives a token from the right, it does the following

```
1. Foreach data part in the left memory
        Begin
        2. Set FAIL = 0.
        3. Foreach variable to test until FAIL = 1
                Begin
                4. Use the first index vector to extract a
                subelement from the left data part.
                5. Use the second index vector to extract a
                subelement from the right data part.
                6. Using the variable type part of both
```

```
                index vectors, choose which test to perform.
                7. Perform the test.
                8. If the test failed, set FAIL = 1.
                End
        9. If FAIL = 0,
                Begin
                10. Set D = the data part of the token
                from the left concatenated with the data
                part of the token from the right.
                11. Build a token using D and the tag part
                of the token that just arrived.
                12. Send the new token to the successors.
                End
        End
13. Halt.
```

3.4.7 The &NOT Node

The &NOT nodes differ from the <NOT> nodes of Chapter 2 in two ways. An &NOT contains one internal memory rather than two, and it understands three rather than two tag types.

When a token arrives from the left, an &NOT node performs the following steps.

```
1. Set COUNT = 0.
2. Foreach data part in the right memory
        Begin
        3. Set FAIL = 0.
        4. Foreach variable to test until FAIL = 1
                Begin
                5. Use the first index vector to extract a
                subelement from the left data part.
                6. Use the second index vector to extract a
                subelement from the right data part.
                7. Using the variable type part of both
                index vectors, choose which test to perform.
                8. Perform the test.
                9. If the test failed, set FAIL = 1.
                End
        10. If FAIL = 0, set COUNT = COUNT + 1.
        End
11. If the token that just arrived is tagged "VALID", store
COUNT and the data part in the left memory; otherwise if the
token is tagged "INVALID", delete an identical data part and
its count.
12. If COUNT = 0, send the token that just arrived to the
successors.
13. Halt.
```

When an &NOT node receives a token from the right, it performs the following steps.

```
1. If the token is tagged "OLD-VALID", halt.
2. If the token is tagged "VALID", set INC = 1; otherwise
set INC = -1.
3. Foreach data part in the left memory
        Begin
        4. Set NEWCOUNT = the count stored with the data part.
        5. Set FAIL = 0.
        6. Foreach variable to test until FAIL = 1
                Begin
                7. Use the first index vector to extract a
                subelement from the left data part.
                8. Use the second index vector to extract a
                subelement from the right data part.
                9. Using the variable type part of both
                index vectors, choose which test to perform.
                10. Perform the test.
                11. If the test failed, set FAIL = 1.
                End
        11. If FAIL = 0,
                Begin
                12. Set NEWCOUNT = NEWCOUNT + INC.
                13. Replace the count in the left memory with
                NEWCOUNT.
                14. If (NEWCOUNT = 0 and INC = -1)
                or (NEWCOUNT = 1 and INC = 1),
                        Begin
                        15. If INC = -1, set TAG = "VALID";
                        otherwise set TAG = "INVALID".
                        16. Build a token using TAG and the
                        data part from the left.
                        17. Send the new token to the successors.
                        End
                End
        End
18. Halt.
```

3.4.8 The &BUS Node

The &BUS node for OPS2 differs from the bus node in Chapter 2 only in being able to generate three tags rather than two.

```
1. If the working memory change was a delete, set TAG = "INVALID";
otherwise, if it was an add, set TAG = "VALID";
otherwise, set TAG = "OLD-VALID".
2. Build a token using TAG and the affected data element.
```

3. Send the token to the successors.
4. Halt.

3.4.9 The &P Node

This node differs significantly from the corresponding node in Chapter 2. When the OPS2 interpreter adds a new instantiation to the conflict set, it maintains the order of the conflict set by inserting the instantiation an appropriate distance from the beginning of the set.

1. Locate and remove an identical instantiation from the conflict set, if one exists.
2. If the token was tagged "INVALID", halt.
3. Locate the first instantiation in the set that is dominated by the new instantiation.
4. Insert the new instantiation at this point.
5. Halt.

3.4.10 Passing Information to the Node Programs

The node programs must be passed three pieces of information before they can begin execution: the description of the node to be interpreted, the token to be tested, and (only the two-input nodes) an indication of whether it was called by its left predecessor or its right predecessor. This section explains how this information is passed to the compiled Lisp functions that serve as node programs.

The description of the node is passed using the function-calling mechanism of Lisp. The names of the various node programs are the names that appear in the first fields of the nodes. When the Lisp function APPLY is called with that name and the rest of the node as its arguments, it locates the node program, binds the values appearing in the other fields to local variables of the node program, and then passes control to the program.

The token and the indication of which predecessor called the node are passed in global variables. The OPS2 interpreter contains a function that keeps track of all the pending nodes plus the information to be passed to them. Each time one node finishes execution, this function chooses another node, binds the token and the identity of the predecessor to the two global variables, and then calls APPLY to pass control to the node program.

4. Analysis of the OPS2 Algorithm

In this chapter an analysis of the OPS2 match algorithm is performed in order to determine how the time and space costs of the algorithm depend upon the number of productions in production memory and the number of data elements in working memory. The analysis determines only the general forms of the dependencies. For example, it shows that, in the best case, the time cost is logarithmic function of the number of productions, but it does not try to determine the cost more precisely than that. The next chapter contains the results of experiments that determined the precise time and space costs for three typical production systems.

In addition to the usual best and worst case effects, this analysis determines the expected effects of production system size. The expected cost analyses assume the production systems are written in the style of the Instructable Production System group [49, 46]. This group is attempting to build production systems much larger than those in use today, and its efforts have included developing programming techniques that are appropriate for these large systems.

4.1 Best and Worst Case Effects

This section contains a conventional, though informal analysis of the algorithm. The first six subsections below describe the best and worst case effects of production memory size on (1) the number of nodes in the network, (2) the amount of space required to store tokens, and (3) the time required to update the conflict set after each working memory change. The next four subsections describe the best and worst case effects of working memory size on (1) the amount of space required to store tokens, and (2) the time required to update the conflict set. The size of working memory does not affect the number of nodes in the network, of course.

All the bounds derived here are sharp. At least one production system is described that reaches each bound.

4.1.1 The Worst Case Effect of PM Size on Network Size

In the worst case the number of nodes in the network will grow linearly with the number of productions in production memory. Each LHS compiles into one &P node, a few one-input nodes for each condition element, and if the LHS contains more than one condition element, a few memory and two-input nodes. Consider the set of nodes needed to perform the match for production Pj. Let Nj be the size of this set. The number of nodes added to the network is usually somewhat less than Nj, because some of the nodes will be in the network before

the LHS is compiled. In no case is the number of nodes added greater than Nj. The total number of nodes in the network is thus bounded by the sum over all productions, Px, of Nx. A production system that achieves this linear bound will be shown in the next subsection.

4.1.2 The Best Case Effect of PM Size on Network Size

Even in the best case the number of nodes in the network is a linear function of the number of productions in production memory. The network grows most slowly when every LHS compiled is identical to a previously compiled LHS. In this case no nodes are added to the network except &P nodes. But since every production adds one &P node to the network, the rate of growth is linear -- one node per production.

Since production systems do not typically grow in this fashion, however, it is worthwhile to consider what happens when productions with distinct LHSs are added to production memory. Since distinct LHSs will necessarily sometimes match different data, their &P nodes cannot be sons of the same node. (Certainly the &P nodes can have common ancestors; the restriction is simply that they cannot be the immediate successors of the same node.) Consequently, if LHSs are distinct, there must be linear growth in the network somewhere in addition to the &P nodes.

The growth does not have to occur in the same part of the network for every production system. If all the LHSs except a constant subset have only one condition element, the number of two-input and memory nodes will remain constant while the number of one-input nodes grows (at a linear rate). If all but a constant subset have more than one condition element, the number of one-input nodes can remain constant while the number of two-input nodes grows. If neither of these is true, then both the one-input nodes and the two-input nodes must grow in number.

Whether the number of memory nodes grows depends upon which of the other two classes of nodes experiences growth. If the number of two-input nodes remains constant, the number of memory nodes must also. If both the one-input and two-input nodes grow in number, then the memory nodes, located between the two, must grow similarly. Finally, since many two-input nodes can share the same one-input node, it might appear that it would be possible for the number of two-input nodes to grow while the number of one-input and memory nodes remained constant. If the number of two-input nodes is to grow indefinitely, however, it is not. Starting with some set of memory nodes, one can define many LHSs by choosing pairs of memory nodes and joining them with a two-input node. But the number of LHSs that can be defined this way is finite; eventually one will find that every possible two-input node is already in the network. To define more LHSs, it will be necessary to add

two-input nodes above those already in the network -- in effect defining LHSs with three condition elements. Adding these nodes requires that memory nodes be added to the network above the existing two-input nodes. Thus memory nodes must be added, albeit perhaps slowly compared to the rate at which two-input nodes are added.[1]

To show that these bounds are sharp (that is, attainable) it is necessary to construct a production system that attains the bounds. It is difficult to construct one that uses only a fixed number of one-input nodes, but easy to construct one that uses a fixed number of two-input nodes. An example of such a production system is a Turing Machine simulator in which the infinite tape is stored in productions, one production for each tape cell. A tape cell production needs only one condition element in its LHS. If cell 18 held a zero, for example:

```
Cell18  ( (Read 18) --> (Cell 18 0) (<DELETE> (Read 18)) ).
```

Because all but a finite number of the tape cells will be blank at any time, it is necessary to include a default production for the blank cells:

```
DEFAULT ( (Read =X)
          -->
          (Cell =X Blank) (<DELETE> (Read =X)) ).
```

The finite state control could also be encoded in productions. The following production, for example, would fire when the machine read a zero in state S1. It would change the state to S3, write $ onto the tape cell, and then move the head to the right.

```
Q0S1    ( (Cell =C 0) (State S1)
          -->
          (<DELETE> (Cell =C 0)(State S1) )
          (State S3) (Read (<+1> =C))
          (<BUILD> ((Read =C) --> (Cell =C $)) ) )
```

The <BUILD> action writes over the previous contents of the current tape cell by adding a new production that responds to requests to read the cell. Since conflict resolution will choose this new production in preference to all older productions with identical LHSs, there is no need to delete the old productions for the cell. The action (Read (<+1> =C)) causes the head to move to the right; it adds one to the number of the current cell and deposits into working memory a request for the contents of that tape cell.

---

[1] All constructions attempted by this writer have resulted in linear or near-linear growth of the number of memory nodes, but no proof has been found that these constructions are optimal.

When execution of the Turing Machine begins, production memory contains the default production, the state transition productions, and the cell productions for a finite portion of the tape. No productions are added to the finite state control during execution, but the number of tape cell productions grows unboundedly. Since the tape cell productions have single condition elements in their LHSs, this results in adding only &P and one-input nodes to the network.

4.1.3 The Worst Case Effect of PM Size on Token Memory

In analyzing the effects of production memory size on time costs, it is necessary to assume that all the productions require comparable amounts of effort to instantiate. The effort required to instantiate the LHSs is relevant because the expenditure of unusually high amounts of effort often results in building a large number of tokens. The assumption of comparable complexity can be made because the effects of complexity are explored in the section on working memory size.

With this assumption, the number of tokens stored in the network is at worst a linear function of the number of productions. Section 4.1.1 showed that the number of memory nodes in the network grows at worst linearly with the number of productions. The assumption that the LHSs are comparably complex guarantees that none of the memories will contain an extraordinarily large number of tokens.

It must still be shown that this bound is sharp, for it could be that most of the memories are empty at any given time, and that adding new memories only adds new empty memories. One way to build a production system reaching the bound is to use negated condition elements. The production system might have two condition elements in every production. All the productions would have the same first condition element. Each would have a different second condition element, and all the second elements would be negated. The network for such a production system has one &NOT node for each production. Since all the productions have the same first condition element, the left inputs of the &NOT nodes all come from the same source. If these were &VEX nodes, there would be only one memory for all the left inputs, but since they are &NOT, each has its own memory. With these private memories in the network, the entry into working memory of an element matching the first condition element results in one token being stored for every production in the system.

Since this example relies on &NOT nodes and the fact that they do not share memories, it is interesting to ask whether the same bound can be reached in a production system containing no negated condition elements. Since &VEX nodes always share memories when possible, in the network for such a production system, two memories never take their inputs from the

same source. Thus copies of the same token can be stored in two different memories only if the token was passed by two different sets of nodes -- or equivalently, if the data part of the token conformed to two different descriptions. It is not difficult for the data to do this; condition elements are templates describing some, but usually not all, of the features of the data they match. Since the data elements -- or at least the part of the data elements that the condition elements describe -- are not infinite, however, only a finite number of features can be chosen to describe the elements. These features can be combined in various ways to generate different partial descriptions, but only finitely many combinations are possible. This argument also applies to the memories holding pairs and longer n-tuples of elements; the n-tuples also have finite limits on the number of partial descriptions to which they conform. If the data part of a token can be described in only a limited number of ways, then there is necessarily a limit to the number of different node memories holding the token. The worst case upper bound must therefore be a constant; production memory can grow arbitrarily large, but after every data element appears in its maximum number of memories, the number of tokens stored will not grow.

Two points about this bound should be noted. First, it requires the assumption that there are no &NOT nodes in the network. The true worst case bound is the linear bound achieved by considering the &NOT nodes. Second, the constant bound is extrememly large. A data element of K subelements can be described in 2^K different ways just by choosing different subsets of constants. With length tests, variables, and user defined predicates, the number of distinct descriptions can be increased greatly. N-tuples of elements can be described in many more ways. It is unlikely that any production system could grow large enough to reach the constant bound.

4.1.4 The Best Case Effect of PM Size on Token Memory

In the best case, production memory can grow without adding to the number of tokens stored. The Turing Machine production system grew indefinitely while adding no memory nodes to the network. When the number of memories does not grow, the number of tokens stored does not grow either.

4.1.5 The Worst Case Effect of PM Size on Time

The worst case time cost, like the two worst case space costs, is a linear function of the size of production memory. Since the only possible interaction between LHSs is beneficial -- a reduction in the effort resulting from shared nodes -- the bound cannot be worse than linear. To see that the linear bound is sharp, look at the Turing Machine production system or the production system in section 4.1.3. In the network for the Turing Machine simulator,

the number of one-input nodes activated on each cycle increased linearly with the number of productions. In the network for the other production system, a similar increase occurred in the number of tests required to update the node memories.

4.1.6 The Best Case Effect of PM Size on Time

As in the discussion of best case network size, unless some assumptions are made about the system, the bounds achieved in this subsection will be uninteresting. If one asks simply, "How fast do the time costs grow as production memory size increases?" then the answer can be that they need not increase at all. A production system could be built which had a fixed core of productions that fire and a growing body that never fire. If the condition elements of the productions that never fired were sufficiently different from the conditions in the others -- perhaps all the conditions in the silent productions having exactly thirteen subelements while none of the others had thirteen -- a single node activation after each action would suffice to process all their conditions, regardless of the number of productions. But a production system containing a large body of productions that never fire is unrealistic. In an attempt to get a more interesting bound, the analysis in the rest of this section will assume that the collection of productions in production memory are all equally likely to fire.[1]

One more assumption will be made to simplify the analysis, but it should have no effect on the results achieved. The assumption is that instantiations enter the conflict set at the same rate that productions fire. Obviously they cannot enter at a slower rate, for the production system would eventually empty the conflict set and halt. If they entered faster, some would never be executed, and the effort expended in finding them would be wasted. In a best case analysis, the assumption should be made that effort is not wasted.

With these two assumptions made, the question to be asked becomes, "How much processing must be performed in order to add an average of one instantiation per cycle to the conflict set, assuming that all productions are equally likely to enter the set?" The general form of this question is, "How difficult is it to select one element from a set if all the elements are equally likely to be selected?" The answer to the general question is that methods which invoive making binary comparisons require on the order of $\log_2(S)$ steps, where S is the size of the set. The best case bound is, therefore, that the time costs grow logarithmically with the size of production memory.

This bound is sharp; the OPS2 match can come within a constant factor of it. Suppose a

---

[1] It would be interesting to perform this analysis assuming that all productions can fire, but making no assumption about their relative likelihoods of firing. This analysis will not be attempted here.

vector is stored in production memory, one production for each of its elements, and suppose the indices of the vector are binary numbers represented as lists of bits:

```
(INDEX 0 1 0 0 . . . 1 1).
```

If element 3 of a 32 element vector held the number 2.71828, the production for this element would be

```
Element3 ( (INDEX 0 0 0 1 1) --> 2.71828 ).
```

An OPS2 network for a production system like this would require $2\log_2(K) + 2$ tests to select an element from a vector containing K elements. One test would be made for the length of the index, one for the constant INDEX, and then each bit of the index would be compared to 0 and to 1. Two tests of each bit would be made because the OPS2 match does not understand mutually exclusive conditions.

This production system does not show exactly what was wanted, however; the production system cannot grow beyond a bound fixed by the length of the index. Indices of B bits suffice to index only 2^B elements. But slightly changing the representation yields a production system that is able to grow indefinitely. The change is to represent the binary numbers as variable length strings of 0's and 1's. The index for element 2 is written (INDEX 1 0); the index for element 14 as (INDEX 1 1 0 1). The production for element 3 of the vector would then be written

```
Element3 ( (INDEX 1 1) --> 2.71828 ).
```

Selecting an element from a vector of K elements using this representation requires approximately $3\log_2(K)$ tests (1.5 times as many as before). On the first level of the network, where tests of element length are made, $\log_2(K)$ tests are performed on each token. The $\log_2(K)$ tests include one for indices of 1 bit, one for indices of 2 bits, and so on up to $\log_2(K)$ bits. $\text{Log}_2(K)$ is the length of the longest of the indices. Half of all the indices are this long. Of the remainder, one-half have $\log_2(K) - 1$ bits, one-fourth have $\log_2(K) - 2$ bits, one-eighth have $\log_2(K) - 3$, and so on. Only two indices have only 1 bit. If all the indices are assumed to be equally likely to occur, then the longer paths will process more tokens than the shorter ones. Thus the number of tests of binary digits approaches the length of the longest path: $\log_2(K)$. Again, two tests are performed on each binary digit, and one test is performed on the constant INDEX. The total number of tests is thus approximately

1	(the constant INDEX)
+ $\log_2(K)$	(the length tests)
+ $2\log_2(K)$	(the tests for C and 1)

4.1.7 The Worst Case Effect of WM Size on Time

When an element is added to or deleted from working memory, a token is created and processed by the match routine. Some number of one-input nodes are activated, and possibly one or more memory and two-input nodes. Each of the one-input nodes performs exactly one test, but the number of tests performed by the two-input and memory nodes depends on the current contents of working memory. When working memory is large, the node memories often contain many tokens, causing the two-input and memory nodes to perform many tests when they are activated. In addition, with their input memories holding many elements, the &VEX nodes are likely to produce several output tokens for each input token. Each of the new tokens will cause further processing. The purpose of this section is to determine an upper bound for the effects of working memory size. Since the effects of production memory size on the time costs have already been discussed, it is sufficient here to consider only a single production.

The number of tests performed by the two-input and memory nodes is a maximum when the memories of the network contain the greatest possible number of entries. Since tests performed by one-input and two-input nodes prevent some tokens from reaching the memories following those nodes, the number of entries in the memories is greatest when no such tests are performed. The worst case, then, is a production whose condition elements have no selectivity:

```
PEXP    ( =X1 =X2 =X3 . . . =Xc --> . . . ).
```

If working memory contains W elements and the production's LHS contains C condition elements, then the two-input nodes for this production perform on the order of W^{C-1} operations in processing a single working memory change.[1] Since the LHS contains C condition elements, the network contains C-1 &VEX nodes. The first of these has W elements in each of its input memories, so when it receives a new token on either input, it produces W output tokens. After each working memory change it receives 2 tokens describing the change (a token is received on each input) and therefore outputs 2W tokens. The second

[1] If the working memory change is an add, working memory's size will become W+1; if the change is a delete, its size will become W-1. Depending on the order in which the nodes are executed, the number of tests performed could be of the order $(W-1)^{C-1}$ or $(W+1)^{C-1}$. But since this section is concerned with large W, the difference is not significant and can be ignored.

&VEX node receives these 2W tokens on its left input, and the 1 token resulting from the working memory change on its right. The left memory of this node holds all pairs of elements matching =X1 and =X2, a total of W^2 elements. Thus the 1 element arriving on the right causes the node to output W^2 tokens. Since this node, like the first, has W elements in its right input memory, the 2W tokens arriving on the left cause $2W^2$ output tokens, for a total of $3W^2$ tokens output. A similar analysis shows that the third &VEX node has W^3 elements in its left memory (all the three-tuples matching =X1, =X2, and =X3) and therefore produces $4W^3$ output tokens. Continuing the analysis in this way shows finally that the last &VEX node (the C-1st) outputs CW^{C-1} tokens. The total number of tokens built and output is

$$CW^{C-1} + (C-1)W^{C-2} + \ldots + 2W.$$

If W is large, the first term will dominate. The time costs can therefore be approximated by CW^{C-1}.

The number of tests performed in updating the node memories can be greater than this. Suppose the oldest element in working memory is deleted. Then the two-input nodes will execute as just described, producing the stated number of output tokens. Tagged INVALID since they result from a delete, these tokens will cause the &MEM nodes to search through their memories and delete identical data parts. Since the working memory element deleted was the oldest in working memory, the data parts to be deleted will be distributed uniformly throughout the node memories (e.g., in the memory following the first &VEX node, the data parts will occupy the last position, the last-C position, the last-2C, and so on). On the average, then, deleting one of these data parts will require searching through just over one-half the elements in the memory. Assuming the number is exactly one-half, each token processed by the &MEM node following the first &VEX node will require $0.5W^2$ tests. Since the first &VEX node outputs 2W tokens, the total cost at this &MEM node is W^3 tests. Each token processed by the &MEM node following the second &VEX node requires $0.5W^3$ tests, for a total of

$$(0.5W^3 * 3W^2) = 1.5W^5.$$

The &MEM node following the third &VEX node performs

$$(0.5W^4 * 4W^3) = 2W^7.$$

In general, the &MEM node following the K^{th} &VEX node performs

$$(0.5W^{K+1} * (K+1)W^K) = 0.5(K+1)W^{2K+1}.$$

The C-2nd &VEX node is the last to be followed by an &MEM node. It performs

$(0.5W^{C-1} * (C-1)W^{C-2}) = 0.5(C-1)W^{2C-3}$.

When W is large this term dominates, and the costs can be approximated by $0.5(C-1)W^{2C-3}$.

Finally, the &P node may perform more operations then either the &VEX or &MEM nodes. Most tokens arriving at PEXP's &P node result in scanning the entire conflict set. All tokens tagged VALID cause the node to scan through the entire set.[1] Tokens tagged INVALID or OLD-VALID cause the node to scan the entire set if they arrive after the old instantiation has been executed. Since the &P node is the successor to the $C-1^{st}$ &VEX node, if working memory holds W elements, then the conflict set may contain W^C instantiations. (It contains fewer only if some of the instantiations have already executed.) Since this is a worst case analysis, assume that the size of the conflict set is approximated by W^C. After each change made to working memory, the $C-1^{st}$ &VEX node produces CW^{C-1} output tokens. The &P node therefore performs about

$(CW^{C-1} * W^C) = CW^{2C-1}$

tests in updating the conflict set.

In summary, in the worst case one working memory change can result in the &VEX nodes performing CW^{C-1} operations, the &MEM nodes performing $0.5(C-1)W^{2C-3}$, and the &P node performing CW^{2C-1}. These are operations of different kinds, and they are not equally expensive to perform. But regardless of their relative costs, if W is large, CW^{2C-1} will dominate the total cost.

4.1.8 The Best Case Effect of WM Size on Time

The time required to execute a production system can be independent of working memory size provided the production system possesses three properties. First, the condition elements must be discriminating enough that no two-input nodes produce combinatorially increasing numbers of output tokens as working memory grows. Second, either the network must contain no memories, or the productions must never delete any working memory elements except the most recent. Third, the conflict set must not grow; instantiations must not enter the set faster than they can be executed.

These three properties are restrictive, but a production system with the properties can be constructed. The following production system performs the Sternberg classification test [54].

---

[1] It is unnecessary to scan the set, but having been taken unchanged from an earlier version of the interpreter, the &P node is not as efficient as it could be. It has never been changed because the conflict set is usually small, and scanning the set therefore inexpensive.

The system is initially given a set of digits, perhaps {2, 3, 8}, and then given probe digits one at a time. After each probe it answers YES if the probe is a member of the set and NO if it is not.

```
PYES    ( (probe =Y =X)(set =X) --> (answer =Y yes) )

PNO     ( (probe =Y =X) - (set =X) --> (answer =Y no) )

PCONT   ( (answer =X =R)
          -->
          (<WRITE> =R)
          (<WRITE> "Ready for the next probe.")
          (probe (<BIND>) (<READ>)) )
```

The set is represented by one working memory element for each member; the set given above would be

```
(set 2)
(set 3)
(set 8).
```

The probes are held in working memory elements with three subelements. The first subelement is the atom **probe**. The second subelement is a unique integer generated to keep the probe elements distinct. The third subelement is the probe itself. The first two productions examine working memory to determine whether the probe is a member of the set. The third production prints the answer, prompts for another probe digit, and builds the probe element from the input. Working memory gains two elements each time a digit is classified, but since the production system has the above three properties, it can run indefinitely without slowing down.

4.1.9 The Worst Case Effect of WM Size on Token Memory

The worst case effects of working memory size on the number of tokens stored have already been seen. In the section on worst case time costs, a production was defined that required storing the greatest possible number of entries in every node memory:

```
PEXP    ( =X1 =X2 =X3 . . . =Xc --> . . . ).
```

The analysis contained in that section showed that the left input memory of the K^{th} &VEX node stored W^K elements, and the right input memory of every node stored W. (W was the number of elements in working memory and C the number of condition elements in the

production.) It also showed that the conflict set would contain as many as W^C instantiations of PEXP -- the instantiations must be counted here because each contains a token. Since PEXP contains C condition elements, its network has C-1 &VEX nodes. The total space cost is therefore

W^C	(the conflict set)
$+ W^{C-1} + \ldots + W$	(the left input memories)
$+ (C-1)W$	(the right input memories)

If W is large the first term dominates this expression, and the space costs can be approximated by W^C.

4.1.10 The Best Case Effect of WM Size on Token Memory

The amount of space required to store tokens can be independent of the size of working memory if the production system meets two requirements. First, the conflict set must not grow; instantiations must be executed as fast as they enter the set. Second, the network must contain no memories.[1] This will be the case only if the productions all have exactly one condition element.

These requirements are more restrictive than the three given in the section on constant time costs. (This should not be surprising since a production system with constant space costs will also certainly have constant time costs.) Because it is interesting to consider whether these more restricted production systems can perform the same tasks, the Sternberg task has been chosen to show that the constant space bound is sharp:

```
PYES1   ( (probe =Y =P =P = =) --> (answer =P yes) )

PYES2   ( (probe =Y =P = =P =) --> (answer =P yes) )

PYES3   ( (probe =Y =P = = =P) --> (answer =P yes) )

PNO     ( (probe =Y =P #P #P #P) --> (answer =P no) )

PCONT   ( (answer =Y =R)
          -->
          (<WRITE> =R)
          (<WRITE> "Ready for the next probe.")
          (probe (<BIND>) (<READ>) 2 3 8) )
```

[1] If there are classes of elements, only some of which increase in numbers, this constraint can be weakened to require only that there be no memories storing elements like those whose numbers are increasing.

The restricted LHS format was compensated for by writing more productions and making them specific to one initial set. Since PCONT contains the initial set of integers in its RHS, it cannot be built until after the initial set is given. The first four productions are also specialized in that they will work only with sets of exactly three elements. They are not so specialized, however, that they would have to be built after the initial set was given; the production system could contain a group of productions for every possible set size.

4.2 Expected Effects

The analyses in this section are very different from the ones in the last section. The outline of each analysis in the previous section was the same; a bound was predicted based on properties of the algorithm or the task in general, and then a production system was described which achieved the bound. In this section, the analysis begins with a production system, or more properly, a class of production systems. An attempt is made in this section to take a set of properties that are common to large production systems, and to determine from these how the costs of interpreting production systems will increase as the systems grow larger.

4.2.1 Characteristics of Large Production Systems

This subsection points out the properties of large production systems that are relevant to the expected cost analyses. As mentioned before, the assumption will be made that the production systems written by the Instructable Production System (IPS) group are typical of large systems. The discussion in this section is one person's interpretation of the experiences of this group in building systems containing a few hundred productions. Since it is an interpretation and since new techniques for writing production systems will undoubtedly emerge as the systems are made bigger, the analysis is at best suggestive, and certainly not to be accepted without question.

Most of the productions written by the IPS group are sensitive to goals. Generally the first condition element in a production matches a goal, and the remaining condition elements locate the data to be processed and test the state of working memory. The productions in the MKYBAN production system are typical. In production MB8, for example, the first condition element matches goals asking for one object to be moved near another, and the remaining two condition elements test whether the object is light enough to carry and whether it needs to be moved from its current location.

```
MB8     ( (Want (=O Near =P)) (Light =O) (=O Near #P)
          -->
          (Want (Monkey Holds =O))  )
```

A common form of growth in these production systems is the implementation of a new goal type. The user creates a new goal class name, like **Holds, On,** or **Near,** and writes a set of productions whose goal condition elements contain the class name.[1] When instances of the goal appear in working memory, these productions fire and attempt to satisfy the goal.

Sometimes the initial set of productions proves inadequate, and then in order to refine the system's response more productions must be added. The productions are commonly added using a technique called renaming. One production is added that recognizes the situation which gave trouble. When it fires, this production transforms the goal in some way; it might, for example, change the goal class name from **On** to **On1**. The rest of the productions that are added are sensitive to the transformed goal; when it enters working memory, they are activated to perform the new actions. The reason for transforming the goal is to prevent interactions with the existing productions. If the new productions were sensitive to the same goals and data elements as the old productions, the old productions would sometimes fire at the wrong time and take control away from the new ones.

Two other forms of growth are possible, but in this writer's opinion neither will be commonly used. The first is adding productions that are not specific to a goal. These productions either contain no goal-matching condition element or contain elements that will match goals from many classes. Production MB10 is an example.

```
MB10    ( (Want =Z) =Z --> (<DELETE> (Want =Z)) )
```

The reason for believing this kind of growth will be rare is simply that in the past productions of this nature have received limited use. The other possible form of growth in production systems is to refine an existing goal without using renaming. This involves simply adding a group of productions that are sensitive to the same elements as the existing productions. The reason for believing this technique will see limited use is that it makes controlling interactions difficult. Because production memory is unstructured, searching through it to determine how the production system would behave in a given situation could be prohibitively time consuming.

---

[1] In many production systems, goal classes are indicated by a conjunction of two or more names. When this practice is followed, it is possible to create a new goal class by combining existing constants rather than adding a new constant. This distinction makes no difference in the analyses that follow.

If it is true that the two most common forms of growth in a large production system are the implementation of new goal classes and the refinement of old ones through renaming, two things will be true of the goal class names:

- New constant names (or new combinations of existing names) will be introduced with most new groups of productions.

- There will be a limit to the number of productions sensitive to each constant name or combination of names.

New constant names will probably also be introduced for the other working memory elements -- the ones holding the data relevant to the goals -- though the rate of introduction may be slower. It is of course possible to add a set of productions that operate on the same data as an existing set; in a production system for designing electronic circuits, for example, many productions may contain data elements that match assertions about transistors. But it must surely be true in general that large production systems have more diverse abilities than smaller ones. As a production system grows and acquires these abilities, new classes of data relevant to these abilities must occasionally be introduced.

This argument does not imply that the average number of productions sensitive to a given data class will be bounded. The argument would allow the number of constants to be a sublinear function of the number of productions in the system. But another argument can be made that the number of productions sensitive to a data class should not increase rapidly. Surely the more there is known about a given set of data, the less likely it is that the need to learn more will arise. Since all information in a production system is encoded procedurally, the number of productions sensitive to a given class of data is a reasonable measure of the amount known about that class. Thus the rate at which productions sensitive to a given data class are acquired is probably inversely related to the number that already exist.

Thus it seems that rather weaker predictions can be made about the data class names than were made about the goal class names:

- New data class names will frequently be introduced with new groups of productions.

- The average number of productions sensitive to a given data class name may increase as the production system grows, but the rate of increase should be a sublinear function of production memory size.

4.2.2 The Expected Effect of PM Size on Network Size

Since the network as a whole grew linearly in both the best and the worst cases, it will grow linearly in all cases. In the best case, however, some kinds of nodes grew in number either very slowly or not at all. A question to consider in this section is whether this will also happen with the typical production systems described above. If the practice of renaming goals is followed as extensively as was suggested above, then the answer to the question is no; linear growth can be expected on all levels of the network. Every new goal type will resuit in new one-input nodes to recognize it. New two-input nodes and new memory nodes will be needed to join these into the network.

4.2.3 The Expected Effect of PM Size on Token Memory

As just argued, the number of memories in the network can be expected to grow linearly with the size of production memory. The question to ask in this section is whether the fraction of the memories that are non-empty remains constant or decreases. Since most of the productions that fire are sensitive to goals, the bulk of the data elements in working memory at any given time are goals and the data that is relevant to the processing of these goals. Most of the tokens held in the node memories therefore contain either goals, data of this kind, or n-tuples of goals and data. Each of these kinds of tokens is considered separately in the following paragraphs.

The number of stored tokens holding goal elements should not increase. Because of the practice of renaming goals, when a new set of productions is added to a production system new goal class names are introduced. This is true whether the productions give the production system a new ability or enhance an old one. Since the goal class names appear in the condition elements matching the goals, the goals can be discriminated by the one-input nodes. The goals for **Move** will not enter the memories of the productions for **Find**, or even **Move2**.

This easy discriminability of goals also has the effect of preventing an increase in the number of tokens for n-tuples of elements. The condition element that matches goals is conventionally written first in a production's LHS. Since the OPS2 compiler arranges the two-input nodes for each LHS in a linear sequence (reflecting the sequence of condition elements in the LHS) no tokens containing more than one data element are constructed unless an appropriate goal is in working memory.

The last group of tokens to be considered are those that hold the data relevant to the processing of goals. An increase can be expected in the number of stored tokens of this

kind, but the rate of increase should be less than linear. As argued before, the rate at which productions sensitive to a given data class are acquired is probably inversely related to the number that already exist. Furthermore, adding a set of productions sensitive to a particular data class does not necessarily increase the number of memories for that class. Existing memories will be shared unless a new condition element contains a unique set of constants and other features tested by one-input nodes. The likelihood of creating a unique set of one-input nodes must decrease as the number of different sets already in use increases.

4.2.4 The Expected Effect of PM Size on Time

In determining the expected dependency of time on production memory size, this section considers separately the effort expended at the one-input nodes, the two-input nodes, the memory nodes, and the &P nodes.

Because of the practice of renaming goals, the effort expended at the one-input nodes increases linearly with the size of production memory. To make it easier to describe the effect, assume that the class of a goal (**Move**, **Find**, **Stack**, etc.) is the first subelement of the goal. Tests for constants in this position are made by the nodes on the second level of the network. Since renaming goals causes the number of classes to increase linearly with the number of productions in the system, the number of nodes on this level will also increase linearly. The nodes preceding these &ATOM nodes perform tests of the length of the goal elements. Since goals do not vary greatly in the number of subelements they contain, this length test will not discriminate among the goals very well. Consequently, the number of &ATOM nodes activated will increase linearly with the number of nodes on the second level.

The effort expended at the memory nodes should increase at a sublinear rate. This follows immediately from the results of the last section; the effort expended at the memory nodes varies directly with the average number of non-empty memories in the network.

The number of two-input nodes activated will increase along with the number of memory nodes, but the total effort expended at the two-input nodes will grow more slowly. This follows from the assumption that new goal class names are introduced along with every new collection of productions. Since the condition elements to match goals are placed first in the LHSs, most of the two-input nodes will have empty left input memories. As the number of node activations increases, the number of nodes with non-empty left memories will remain nearly constant. Thus an ever growing fraction of the nodes will do nothing except examine their memories to find that they are empty.

If the number of two-input nodes with non-empty left memories does not increase, then the number of &P nodes activated should not increase either. Thus the effort expended at

the &P nodes should be nearly constant.

4.2.5 The Expected Effect of WM Size on Time

The expected effect of working memory size on the time required to execute a production system cannot be predicted. The production system characteristics that determine the effects on time are a matter of style. Some programmers consistently write productions that are very sensitive to the size of working memory, and some never write such productions. Some programmers mix the two, writing some productions that are sensitive to the size and others that are not. The production systems written by those in this last group vary in their time costs from one cycle to the next as different goals are deposited into working memory.

4.2.6 The Expected Effect of WM Size on Token Memory

The expected effect of working memory size on the space required to store tokens can be determined little more precisely than the expected effect on time costs. The only difference here is that the range of possible costs can be narrowed somewhat. The time costs vary from constant (i.e., no effect) to polynomial. The degree of the polynomial is the number of condition elements in the production. The space costs range from linear to polynomial. The reason for the new lower bound is that since single-condition productions are rare, most working memory elements have images in at least one node memory.

4.3 Summary of Costs

Tables 4.1 and 4.2 summarize the results of this chapter. The usual notation for complexity (see [1]) is used: $O(1)$ indicates constant cost; $O(x)$ indicates the cost is a linear function of x; $O(f(x))$ indicates the cost varies as the function f.

	Best	Worst	Expected
Network size	$O(P)$	$O(P)$	$O(P)$
Token Memory	$O(1)$	$O(P)$	$<O(P)$
Time	$O(\log_2(P))$	$O(P)$	$O(P)$

P is the number of elements in production memory.

Table 4.1. PM Size Effects

	Best	Worst	Expected
Token Memory	$O(1)$	$O(W^C)$	?
Time	$O(1)$	$O(W^{2C-1})$	?

W is the number of elements in working memory.
C is the number of condition elements in the productions.

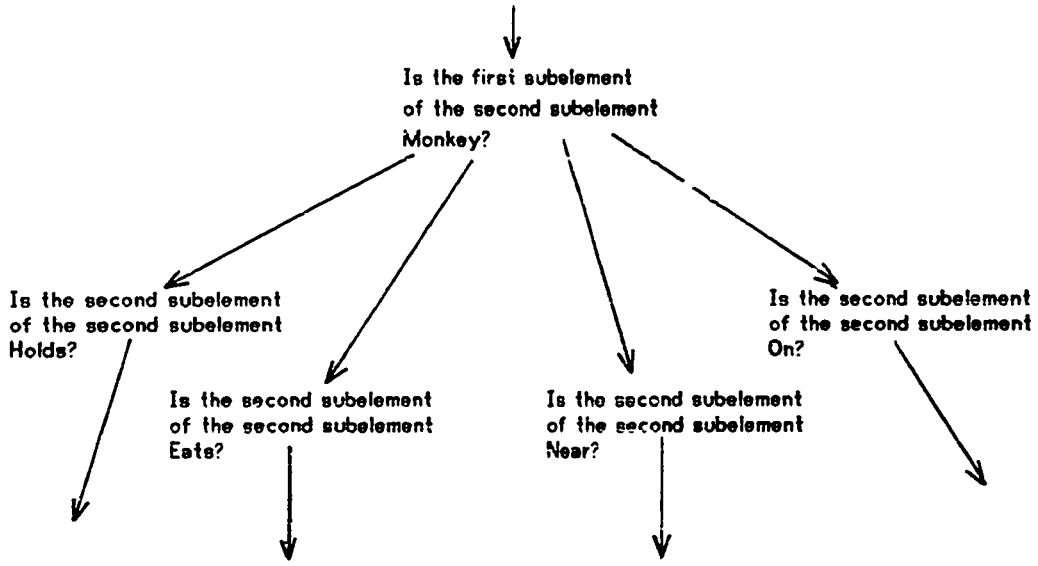
Table 4.2. WM Size Effects

4.4 Improving the Performance of the OPS2 Match

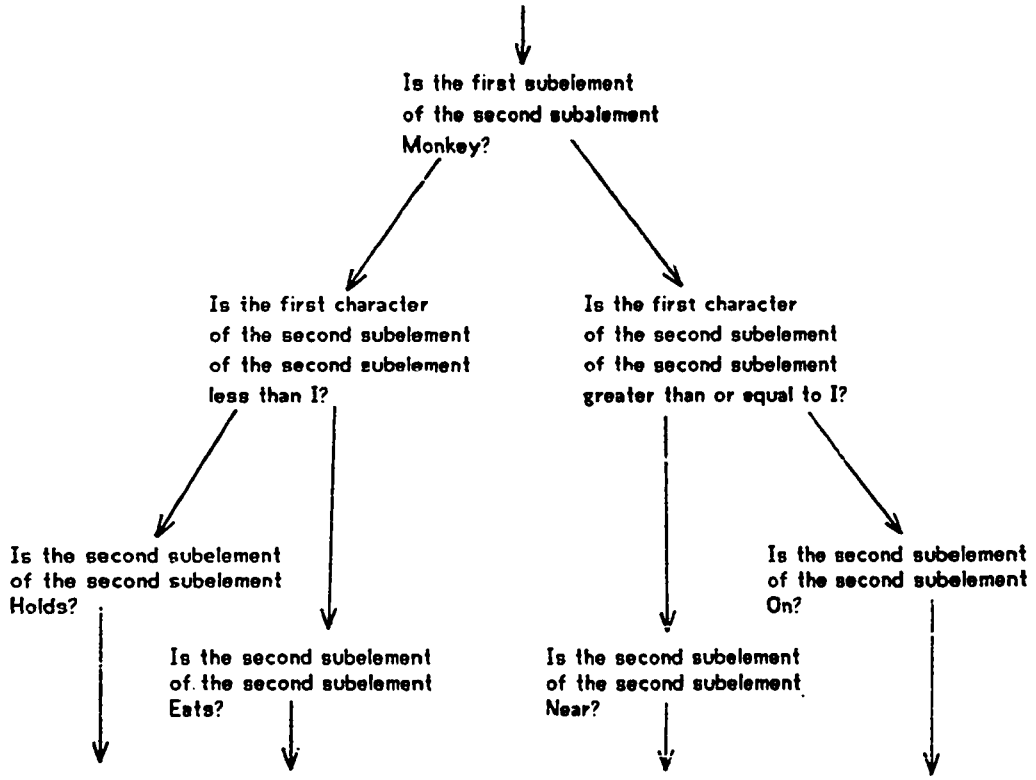
The analysis in this chapter has uncovered a significant weakness in the OPS2 match algorithm, namely the expected time complexity of $O(P)$. However, this is the result of the particular implementation of the Rete Match Algorithm, not an inherent property of the algorithm. This section describes two minor modifications, either of which would reduce the complexity.

4.4.1 Binary Search

The linear dependency arises because of the particular set of one-input nodes chosen. Use of these nodes sometimes causes the match to scan linearly through long lists of mutually exclusive constants. For example, in the case of goal class names mentioned before, the network might include the following set of nodes



Since these are mutually exclusive constants, a binary search could be used, reducing the effort to a logarithmic function of the set size. To incorporate such a search into the network, a new node type testing a lexicographic order on the constants could be used. In practice, this might be implemented by the interpreter's replacing each constant by a number, converting back to a string of letters only to communicate with the user. But to make the example more readable, assume the order is based on the first character of the names. Then the above part of the network could be changed to



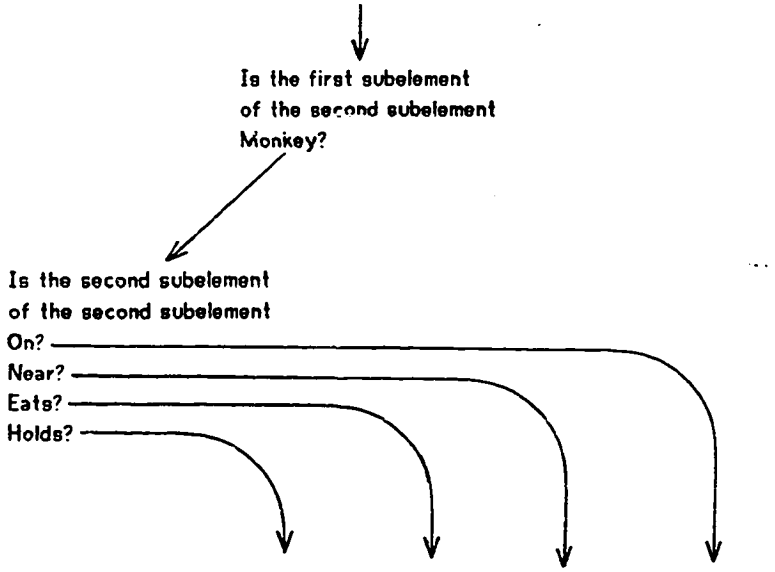
The new nodes are redundant; they would be used only at places in the graph where the cost of a linear search through the successors of a node would be excessive.

There is reason to believe that changing the interpreter in this way would reduce the expected time costs from $O(P)$ to $O(\log_2(P))$. Section 4.2.4 argued that the dominant cost factor in interpreting very large production systems will be the time required to execute the one-input nodes. Section 4.1.6 showed that the OPS2 match can achieve logarithmic time costs if the one-input nodes are arranged appropriately. The OPS2 compiler does not arrange the nodes appropriately for the expected kind of production systems, but the changes suggested here should eliminate its problems.

4.4.2 Hashing

The scheme just described has one unfortunate property: the extra nodes increase the size of the network. An alternative will be described now which will probably not affect the size of the network noticeably. (This depends upon the representation of the nodes, of course.)

Again a new node type must be defined, but this node is quite different from the nodes described in Chapter 2. This node would test multiple, mutually exclusive features. It would distinguish among its output edges, and depending on which feature a given token possessed, it would select some subset of the edges to receive the token. The network fragment in the last subsection would become



This node would hold the features in a data structure called a hash table. A hash table has the property that the number of entries which must be examined to determine whether a given element is present is independent of the size of the table; if the table is used properly, the average number of elements examined will be less than 2 (see [28]). Thus networks incorporating these nodes are potentially quite fast.

A problem with these new nodes is that they complicate the compiler. If the language allows productions to be added while the system is running (as OPS2 does), then it will happen on occasion that a hash table will be filled. When this occurs, the compiler must build a new node with a bigger table, copy the information from the original node, and link the new node into the network in place of the old one.

5. Measurements of the OPS2 Interpreter

The results of an empirical analysis of the OPS2 interpreter are presented in this chapter. Most of the measurements were made in order to complement the analysis of the previous chapter. These measurements determined the absolute time and space costs of interpreting three of the largest OPS2 production systems. The measurements showed that the effects of growth in the production systems are essentially as Chapter 4 suggested. One additional set of measurements was made in order to determine the potential for parallel execution of the algorithm.

5.1 Measures of Time Complexity

The choice of what to count in measuring the OPS2 interpreter is not obvious. It would be easy to choose an inappropriate measure and give a distorted picture of the complexity. For example, one measure that might seem appropriate is a count of the number of tests performed during the match. As Chapter 4 showed, however, some of the selectivity in the match comes from activating two-input nodes with empty input memories. If only the tests were counted, this effect would be missed, and its suspected increase in importance with larger production memories would go untested.

A conservative approach to measurement was taken in this chapter; an attempt was made to account for all the increase in execution time resulting from larger production memories. (Production memory size is the most interesting independent variable.) The interpreter was divided into functional units whose costs were independent of production memory size, and then during the measured runs, counts were made of the number of times these units were executed. Multiplying the counts by the constant costs of the units and summing should account for all the time. If the time computed this way agrees with the measured times, then no effects of production memory size can have been overlooked.

Ideally, the functional units would have been chosen so that their costs were constant over all production systems, but this was not practical in all cases. Since the programs to perform the act and conflict resolution phases have not been described, their parts could not be so divided. Some of the node programs could not be divided because of the way they were written. Where the division of programs was not possible, the time costs of the programs were measured individually for each production system.

5.1.1 Measures of Time Complexity: Detail

The total time expended on each recognize-act cycle is the sum of two components, a fixed overhead for each cycle and and a cost that varies with number of actions performed by the

production. The fixed overhead includes the time necessary to remove the first instantiation from the conflict set and prepare it for execution, to gather statistics about the execution of the production system, and to print trace information if the user has requested it. Unless an unusually great quantity of trace information has been requested, this overhead is quite small. The per action costs include the time to instantiate the action and the time to perform the match; the match in OPS2 is performed once for each change made to working memory.

5.1.2 The Match

The time cost of performing the match is the sum of the costs of fetching and activating the nodes plus the costs of performing the tests at the nodes. The cost of fetching and activating a node is constant. The cost of performing a test at a one-input or two-input node is constant except for the time taken by INDEX. (INDEX is the function that interprets index vectors and extracts subelements from tokens' data parts.) Thus if the time taken by INDEX is separated out, node activations and node tests can be used as measures of time complexity. But a decision still has to be made about how to handle INDEX.

INDEX has a non-constant cost because it locates subelements by counting. If it is looking for the third subelement of the fifth subelement of a list, it counts until it reaches the fifth subelement, descends into it and counts again until it reaches the third. The average distance searched by INDEX depends on the lengths of the data elements, the positions of the constants in the condition elements, the lengths of the LHSs, and other idiosyncrasies of the program. The average distance should not, however, depend on the sizes of production and working memories. If the data elements, condition elements, and LHSs written by a programmer are all similar, then adding more of these to the system should leave the average distance unchanged. Thus the cost of INDEX can be measured for each programmer and used as a complexity measure. Since the cost of INDEX at the two-input nodes depends on a factor that is irrelevant to the one-input nodes -- the average length of the LHSs -- it is necessary to perform one measurement for the one-input nodes and another for the two-input nodes.

Another program whose time is measured for each programmer is the &P node program. Because the &P nodes use Lisp functions to search through the conflict set and to remove instantiations from the set, the number of tests performed by the nodes could not be counted. Since this was not noticed until after most of the timing runs had been made, correcting the omission would have required repeating many expensive computer runs. This rerun cost might have been deemed acceptable except that the &P node is rather uninteresting -- it is inefficient, performing many more tests than are necessary, and it still accounts for only a small fraction of the total run time. Moreover, the results of the

experiments indicated that the the conflict set did not grow significantly as production memory grew, and so the cost for &P probably remained nearly constant.[1]

In order to make the measurements parsimonious, the complexity measures will be grouped into classes. Activations of &VEX and &NOT nodes will be counted together. All activations of one-input nodes will be counted together. Activations of &TWO and &MEM nodes will be counted together. The tests performed by the &NOT nodes will be counted with the tests performed by two other node types: the &VEX nodes (tests of data) and the &MEM nodes (tests for updating the node memories). The cost of the tests performed by the one-input nodes (excluding the cost of INDEX) can be included in the cost of activating the node since each one-input node performs exactly one test when activated.

Finally, then, the time cost of performing the match is approximated by

$$N1*(Cn1 + I1) + N2*Cn2 + T2*(Ct2 + 2*I2) + Nm*Cnm + Tm*Ctm + Np*Cnp$$

where

N1	is the number of activations of one-input nodes.
Cn1	is the cost of activating a one-input node (including the cost of the test performed by the node).
I1	is the cost of INDEX for the one-input nodes.
N2	is the number of activations of two-input nodes.
Cn2	is the cost of activating a two-input node.
T2	is the number of tests performed by the two-input nodes (excluding the tests for updating memories in &NOT nodes).
Ct2	is the cost of performing one test at a two-input node.
I2	is the cost of INDEX for the two-input nodes.
Nm	is the number of activations of memory nodes.
Cnm	is the cost of activating a memory node.
Tm	is the number of tests performed by the memory nodes (including the tests for updating memories in &NOT nodes).

[1] A fairly narrow bound can be put on the number of tests performed by &P. The current implementation performs at least one test of every element in the conflict set each time it is activated. Under some circumstances, the number of tests performed will approach twice the number of elements in the set. The node program could be rewritten so that it never performed more than one test of each element in the set; the range would then be from one test to the number of elements in the set. Thus the best case in the current implementation is no better than the worst case in the other implementation.

Ctm	is the cost of performing one test at a memory node.
Np	is the number of activations of &P nodes.
Cnp	is the cost of activating an &P node.

5.1.3 The Overhead of the Cycle

As stated earlier, the overhead of the recognize-act cycle includes the time to remove the first instantiation from the conflict set and prepare it for execution, to gather statistics, and to print the requested trace information. Since tracing was disabled during the experiments, all the production systems had the same overhead. Thus the overhead is another factor to be measured once. It will be called O.

5.1.4 The RHS Actions

Performing an RHS action involves two steps. First the action must be evaluated to build the element that is to be added to or deleted from working memory, and then working memory must be examined to determine whether the element is already present.

The time required to evaluate an action depends on the number of RHS functions that are called, the amount of processing performed by the functions, and the number of list cells copied in building the data element. Collectively these might be called the complexity of the action. Since different programmers often use actions of very different complexity, the cost of evaluating the actions is not constant over different programmers. Since the complexity is not affected by the size of production memory or the size of working memory, the cost can be considered constant for a given programmer. Thus the cost of this step is another factor to be measured for each production system.

Examining working memory to determine whether an element is present takes an amount of time that varies with the size of the memory. Working memory is organized as a hash table with linear buckets. Searching for an element involves computing the hash address, retrieving the bucket, and then scanning the bucket for the element. The hash function is simple, and it does not work equally well for all programs. Hence, the time to perform this search depends on both the size of working memory and the style of programming. The time does not, however, depend on the size of production memory. This, then, is another factor to be measured for each production system.

Since the costs of both parts are constant for a given production system, they can be combined into a single measure of complexity. This measure will be called E, the cost of

evaluating a RHS action.

5.1.5 Automatic Deletions

The OPS2 interpreter contains a facility for automatically deleting old elements from working memory. Before a run begins, the user sets a maximum age for the elements (measured in the number of actions that have been performed since the element was added to working memory). The interpreter then examines working memory at intervals during the run, and when it finds elements that are older than the maximum, it deletes them on the next cycle. The frequency of examining working memory depends on the rate at which actions are being performed, so the use of automatic deletion in effect adds a constant overhead to each action. Automatic deletion can be accounted for by adding to the value of E.

5.1.6 The Complete Time Cost

To summarize the results of the previous sections, if the average production has *A* action elements, the total time required for each recognize-act cycle (call this *Tc*) is given by the following equation.

$$Tc = O + A*[E + N1*(Cn1 + I1) + N2*Cn2 + T2*(Ct2 + 2*I2) + Nm*Cnm + Tm*Ctm + Np*Cnp]$$

Production system execution rate is often measured in the number of actions performed each second, rather than the number of productions fired. The total time required for each action, which will be called *Ta*, can be computed by dividing Tc by A.

$$Ta = O/A + E + N1*(Cn1 + I1) + N2*Cn2 + T2*(Ct2 + 2*I2) + Nm*Cnm + Tm*Ctm + Np*Cnp$$

Since this form of the cost formula is somewhat simpler, requiring no nested parentheses, it will be used in the rest of this chapter.

The meanings of these terms again:

Tc	is the total time required for each recognize-act cycle.
Ta	is the total time required to process one WM change.
O	is the overhead of the recognize-act cycle.
A	is the average number of action elements per production.
E	is the average time required to evaluate an action element.

N1	is the number of activations of one-input nodes.
Cn1	is the cost of activating a one-input node (including the cost of the test performed by the node).
I1	is the cost of INDEX for the one-input nodes.
N2	is the number of activations of two-input nodes.
Cn2	is the cost of activating a two-input node.
T2	is the number of tests performed by the two-input nodes (excluding the tests for updating memories in &NOT nodes).
Ct2	is the cost of performing one test at a two-input node.
I2	is the cost of INDEX for the two-input nodes.
Nm	is the number of activations of memory nodes.
Cnm	is the cost of activating a memory node.
Tm	is the number of tests performed by the memory nodes (including the tests for updating memories in &NOT nodes).
Ctm	is the cost of performing one test at a memory node.
Np	is the number of activations of &P nodes.
Cnp	is the cost of activating an &P node.

5.1.7 Implementation Dependent Time Costs

Six factors in the cost formula depend only on the implementation of the OPS2 interpreter: the overhead of the recognize-act cycle (O); the cost of activating a one-input node (Cn1); the cost of activating a two-input node (Cn2); the cost of performing a test at a two-input node (Ct2); the cost of activating a memory node (Cnm); and the cost of performing a test at a memory node (Ctm). The values of these factors have been determined by measurements of the OPS2 interpreter[1] running on a KL version of a PDP-10. The following table contains the results of the measurements.

[1] The interpreter for OPS2 has existed in many versions. The one tested was Version 2.0.

	Cost
0	430 μsec
Cn1	86 μsec
Cn2	130 μsec
Ct2	110 μsec
Cnm	130 μsec
Ctm	58 μsec

Table 5.1. Implementation Dependent Times

5.2 Measures of Space Complexity

Before measuring the space costs, it is necessary to choose a set of complexity measures and determine the values of some constants just as it was for the time costs. These preliminaries are much easier for the space costs, however, because there is less opportunity for error; the act of making measurements cannot perturb the costs as often happens when measuring time costs; and since the space costs are more understandable, it is less likely that any important effects will be overlooked.

To store and interpret a production system requires space (1) to store the nodes, (2) to store the table of index vectors for the nodes, (3) to store the RHSs of the productions, (4) to store the data elements in working memory, (5) to store the tokens in the node memories, and (6) to store instantiations in the conflict set. Only four of these will be considered in the measurements of space complexity: the space for the nodes, the space for the index vectors, the space for the tokens, and the space for the conflict set. The space for the RHSs and for data elements is not considered because OPS2 does not compile either of these; every production and every data element added to the system causes one more list (the RHS or the data element) to be stored.

Because the different node types have different lengths, they cannot be counted together. The division of nodes into one-input, two-input, memory, and &P used for the time costs can be used for space costs also. Although the &USER and &VIN nodes are longer than the other one-input nodes, they occur very rarely, and can be ignored. The &NOT node is one word longer than the &VEX, but this is a difference of less than ten percent.

Seven measures of space complexity are thus needed. The following table lists the number of 36 bit words required to store each. &ATOM nodes were assumed to be typical of one-input nodes. &VEX nodes were assumed to be typical of two-input nodes. &MEM nodes were assumed to be typical of memory nodes. The node costs include one word to link the node to its predecessor. As explained in Chapter 3, the cost of storing a token depends on whether it is stored by an &MEM node or an &NOT node. The value given here assumes that most are stored in &MEM nodes.

	Length (Words)
One-input Nodes	5
Two-input Nodes	9 + 2 for each variable checked
Memory Nodes	6
&P Nodes	6
Tokens	2
Instantiations	8 + 1 for each condition element
Index Vectors	1 + 1 for each index

Table 5.2. Implementation Dependent Sizes

5.3 The Production Systems

Three production systems were used in the experiments. This section describes the production systems and gives the values of the production system dependent factors from the time and space cost formulae.

5.3.1 Description of the Production Systems

The primary criterion used to select the production systems was size; they were three of the largest OPS2 production systems available at the time the experiments were performed (the summer of 1978).

KERNL1, written by Michael Rychener, is the largest production system to result from the work of the Instructable Production System group [49]. This production system contains

about 380 productions. Approximately 260 of these implement a natural language front end for the system. The front end translates instructions given in a restricted subset of English into productions. Another 120 productions were added to the system by the natural language front end. These 120 productions give the system the ability to perform simple tasks in a blocks world-like environment. Using them, the system can move objects from place to place, locate objects fitting a given description, compare two objects to determine their similarities and differences, and perform a handful of other tasks of similar complexity. This production system is particularly appropriate for this study because it makes use of the programming techniques that the group thinks will be used in very large production systems.

HAUNT, written by John Laird of Carnegie-Mellon University, is the largest OPS2 production system, containing 1017 productions. HAUNT is an interactive game for one player. The player instructs the production system to move through a large house in order to accomplish a set of tasks while avoiding pitfalls hidden throughout the house. After each instruction the production system describes the effects of the instruction on the state of the game. The user initially knows neither the tasks to perform nor the location of the pitfalls; he is supposed to learn these things through repeated attempts at the game.

PH-632, which contains 316 productions, models a skilled physicist solving textbook problems in mechanics. It was written by John McDermott of Carnegie-Mellon University and Jill H. Larkin of the University of California at Berkeley to conform to a psychological model developed by Dr. Larkin [35]. When PH-632 begins execution, it is supplied with a symbolic encoding of a problem. The symbolic encoding contains the information that a natural language understanding component could extract from the statement of a textbook physics problem, plus any information given in accompanying pictures. From this, PH-632 constructs a qualitative representation of the problem containing the entities with which mechanics is concerned (forces, energies, and the like). It analyzes the qualitative representation to generate the equations for the problem, and then solves these to get the answer.

5.3.2 Characterization of the Production Systems

Two numbers commonly used to indicate the size of a production system are the number of productions in production memory and the number of data elements in working memory. These numbers alone comprise a rather crude measure, for some productions contain more condition and data elements than others, and some data elements contain more subelements than others. The following is a set of measures that together indicate the size of a production system more precisely.

- The number of elements in working memory. In some production systems this number varies over a wide range. When this is true, either a range, an average,

or both should be given.

- The average length of a working memory element. OPS2 data elements can contain both constant atoms and lists. Both of these should be counted.

- The number of productions.

- The average number of condition elements per production.

- The average number of features per condition element. This is computed by counting the number of tests needed to show that a data element can instantiate the condition element. Because of the LHS functions, the absolute size of the condition element is not useful.

- The average number of actions per production.

A measure of action element complexity would also be useful, but it is difficult to formulate such a measure for OPS2. Because the RHS actions contain function calls, the length of an action element can be very different from the length of the data element it produces. Because the RHS actions are not compiled, no interpreter-dependent measure of the complexity (like the one for condition element complexity) is possible.

The following table contains the values of these measures for the three production systems. The number of action elements per production was obtained from the experiments performed to measure the effects of production memory size; the number of actions performed was divided by the number of productions fired. The values shown for average data element length were obtained by examining working memory at the conclusion of these experiments. The number of data elements is the average size of working memory during the runs. The rest of the numbers were obtained by static measurements of the production systems.

	KERNL1	HAUNT	PH-632
Number of Productions	381	1017	315
Condition Elements per Production	2.1	2.2	4.2
Features per Condition Element	7.6	3.5	7.0
Action Elements per Production	2.2	2.1	3.5
Number of Data Elements	120	75	51
Length of Data Elements	8.6 atoms 1.2 lists	2.9 atoms 0.06 lists	6.6 atoms 1.3 lists

Table 5.3. The Measured Production Systems

5.3.3 Production System Dependent Time Costs

Four of the factors in the time cost formula depend on the production system being run: the average cost of INDEX for the one-input nodes (I1); the average cost of INDEX for the two-input nodes (I2); the average cost of executing an &P node (Cnp); and the average cost of executing an action element (E). The table below contains the values of these factors for the three production systems.

	KERNL1	HAUNT	PH-632
I1	260 μsec	230 μsec	240 μsec
I2	360 μsec	280 μsec	390 μsec
Cnp	2000 μsec	1100 μsec	1700 μsec
E	7100 μsec	2500 μsec	2700 μsec

Table 5.4. Production System Dependent Times

5.3.4 Production System Dependent Space Costs

Three space cost factors depend on the style in which a production system is written. The length of the two-input nodes depends on the average number of variables tested by the nodes. The length of the instantiations depends on the number of condition elements in the productions' LHSs. The length of the index vectors depends on the depth of nesting of the subelements tested. The following table lists the values of these factors for the three production systems. The number of variables tested by the two-input nodes and the average length of the index vectors were determined by direct measurements. The length of the instantiations was estimated from the average number of condition elements per production (from table 5.4). The KERNL1 network averages 0.9 tests per two-input node, the HAUNT network 0.09 tests, and the PH-632 network 1.6 tests.

	KERNL1	HAUNT	PH-632
Two-input Nodes	11	9	12
Instantiations	10	10	12
Index Vectors	4	4	4

Table 5.5. Lengths of Units (Words)

5.4 Measuring the Effects of PM and WM Sizes

This section describes the experiments that were performed in order to determine the effects of production system size on time and space costs.

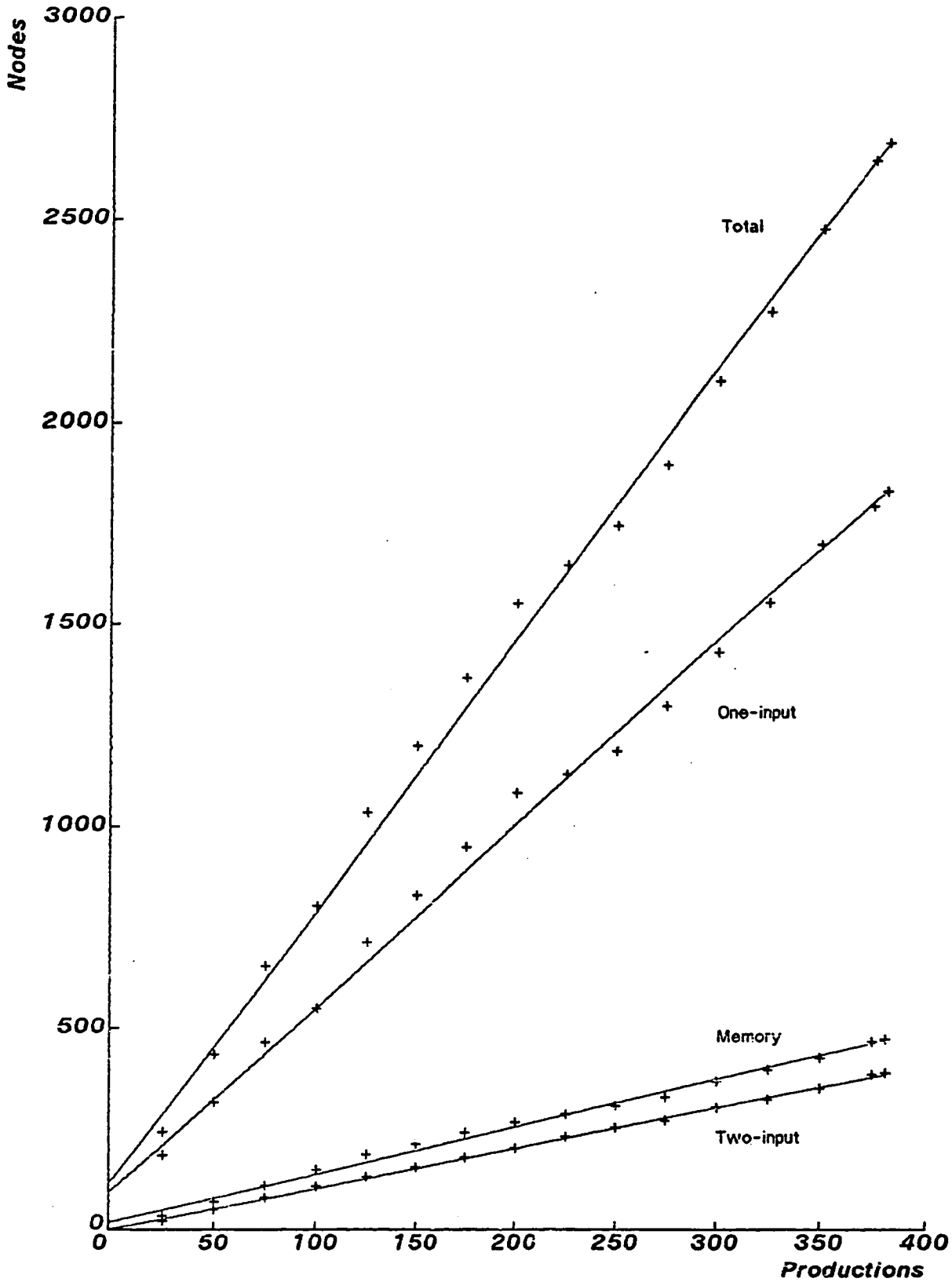
5.4.1 The Effect of PM Size on Network Size

The experiment to determine the effects of production memory size on network size involved compiling the three production systems incrementally. The interpreter was modified so that it would pause after every twenty-five productions to count the number of nodes and index vectors in the network. Graphs 5.1 through 5.6 show the results of the experiments. From these graphs it can be seen that the number of nodes grows at an essentially linear rate, as predicted in Chapter 4. The irregularities in the curves can be attributed to the fact that the experiments did not take into account the existence of groups of related productions. When a group containing unusually complex productions is read in, the network grows faster than usual; when a group containing unusually simple productions is read in, it grows slower than usual. In addition, because productions from the same group are more alike than productions from different groups, the network tends to grow faster when small groups are being read in.

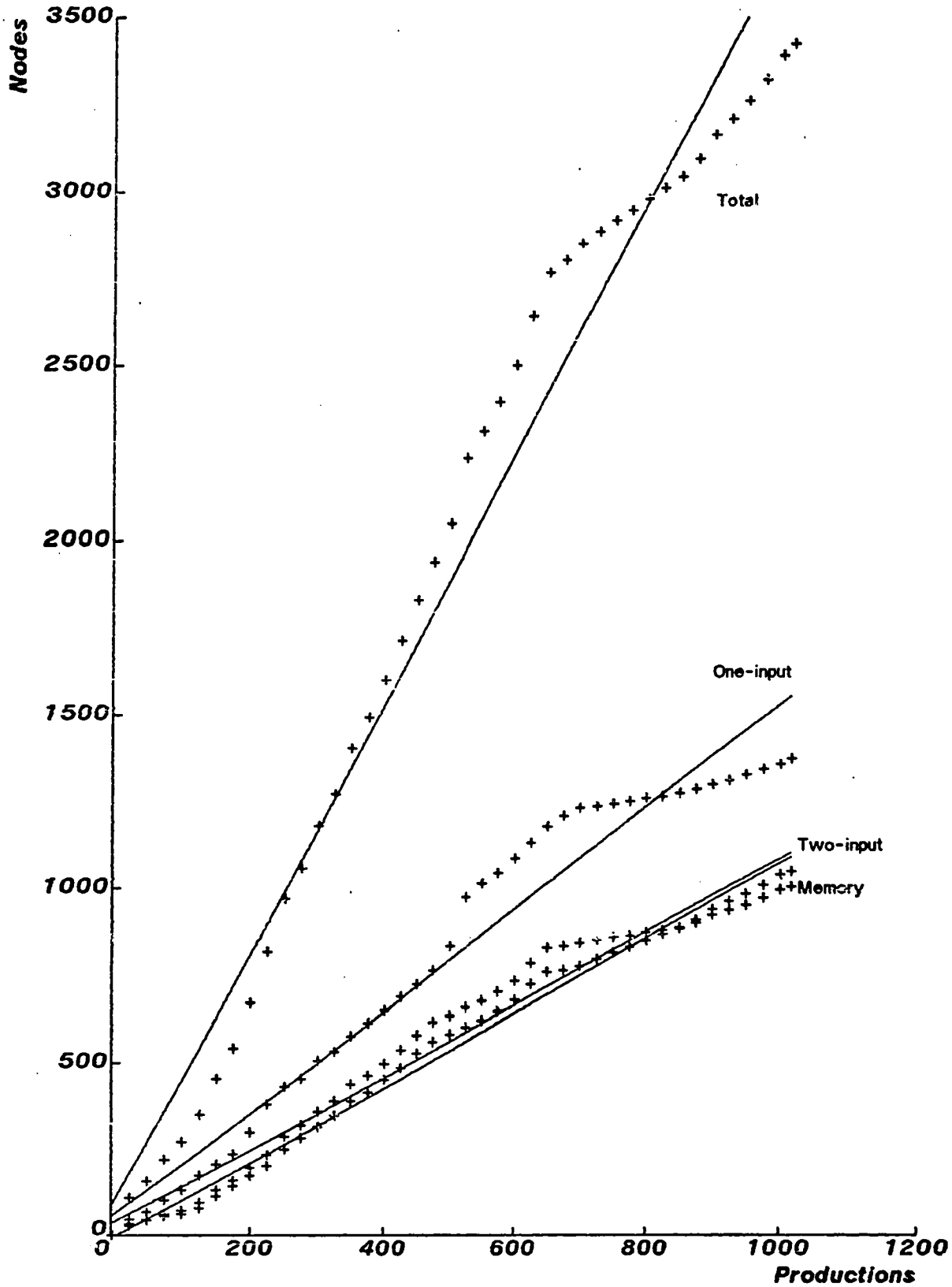
5.4.2 The Effects of PM Size on Token Memory and Time

The experiments to determine the effects of production memory size on token memory and time costs were slightly different for the three production systems.

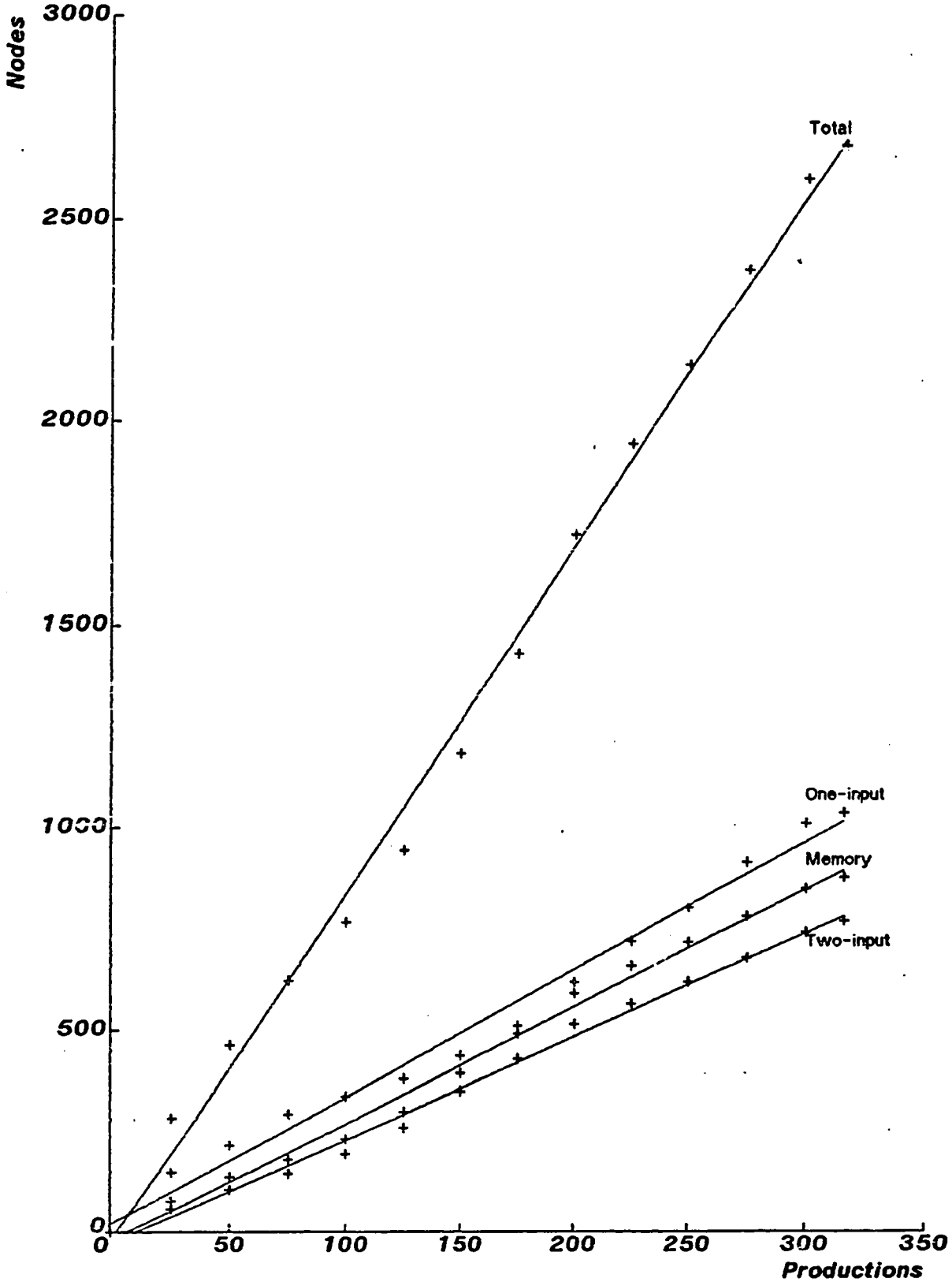
KERNL1 was measured by choosing a task (comparing the descriptions of two objects) that required only 54 productions to perform. The task was performed eleven times, each time with a different number of productions in the system. The largest production memory tested contained 346 productions; larger memories could not be used because adding more productions caused the system to behave differently. On each run the system was allowed to execute long enough for the size of working memory to stabilize (at approximately 120 elements) before the measurements began. During the measured part of the run, 811 productions fired, performing 1784 actions. The distribution of production memory sizes is not uniform because an attempt was made to localize all abrupt changes in the costs. Initially only five runs were made. The first run involved only the 54 necessary productions; on each subsequent run about 70 productions were added to the system. When a sharp increase was found in one of the curves, another run was made with a production memory of intermediate size. This was repeated until the effect could be attributed to a set of productions for a



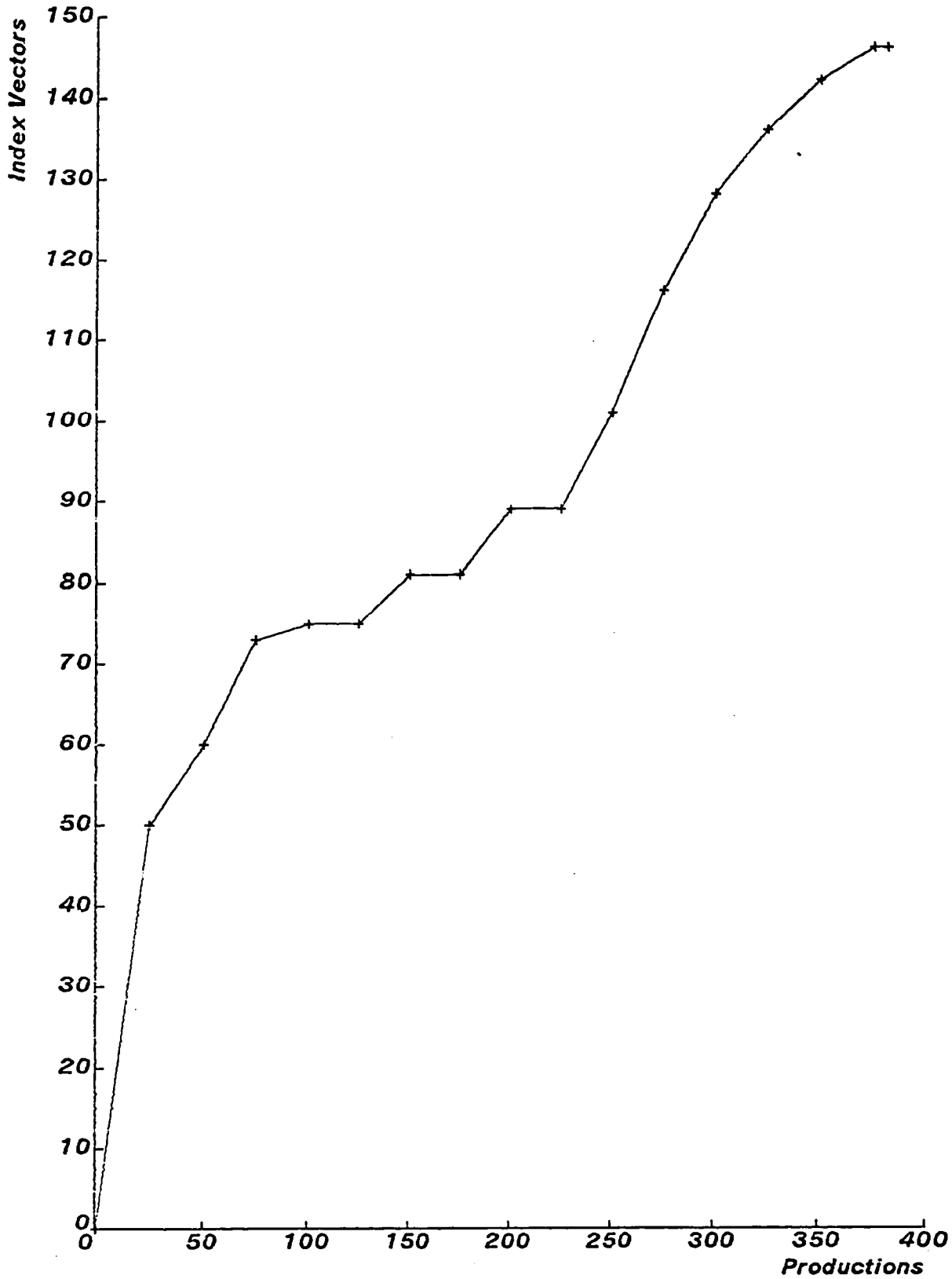
Graph 5.1. Rate of Network Growth: KERNL1



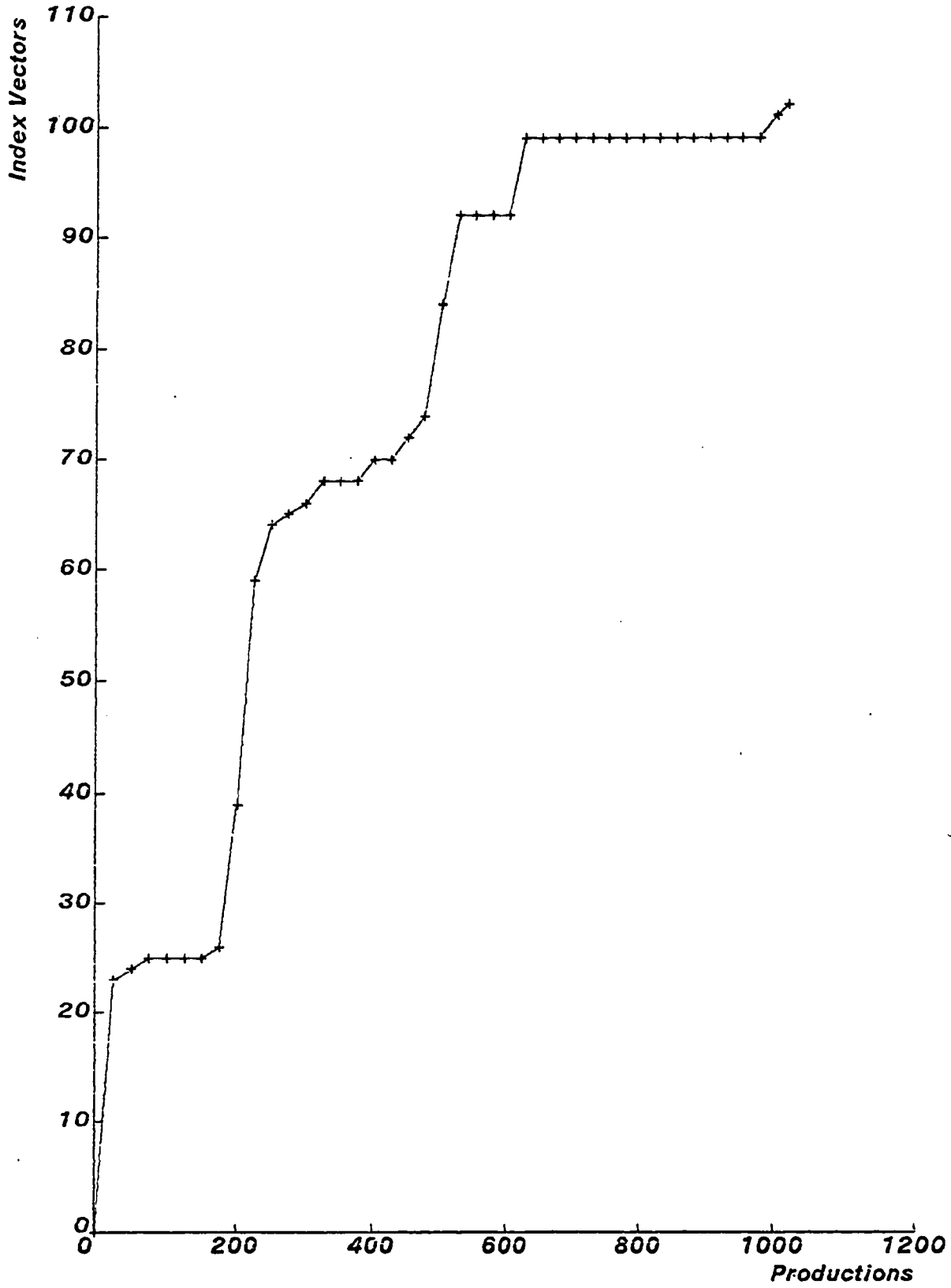
Graph 5.2. Rate of Network Growth: HAUNT



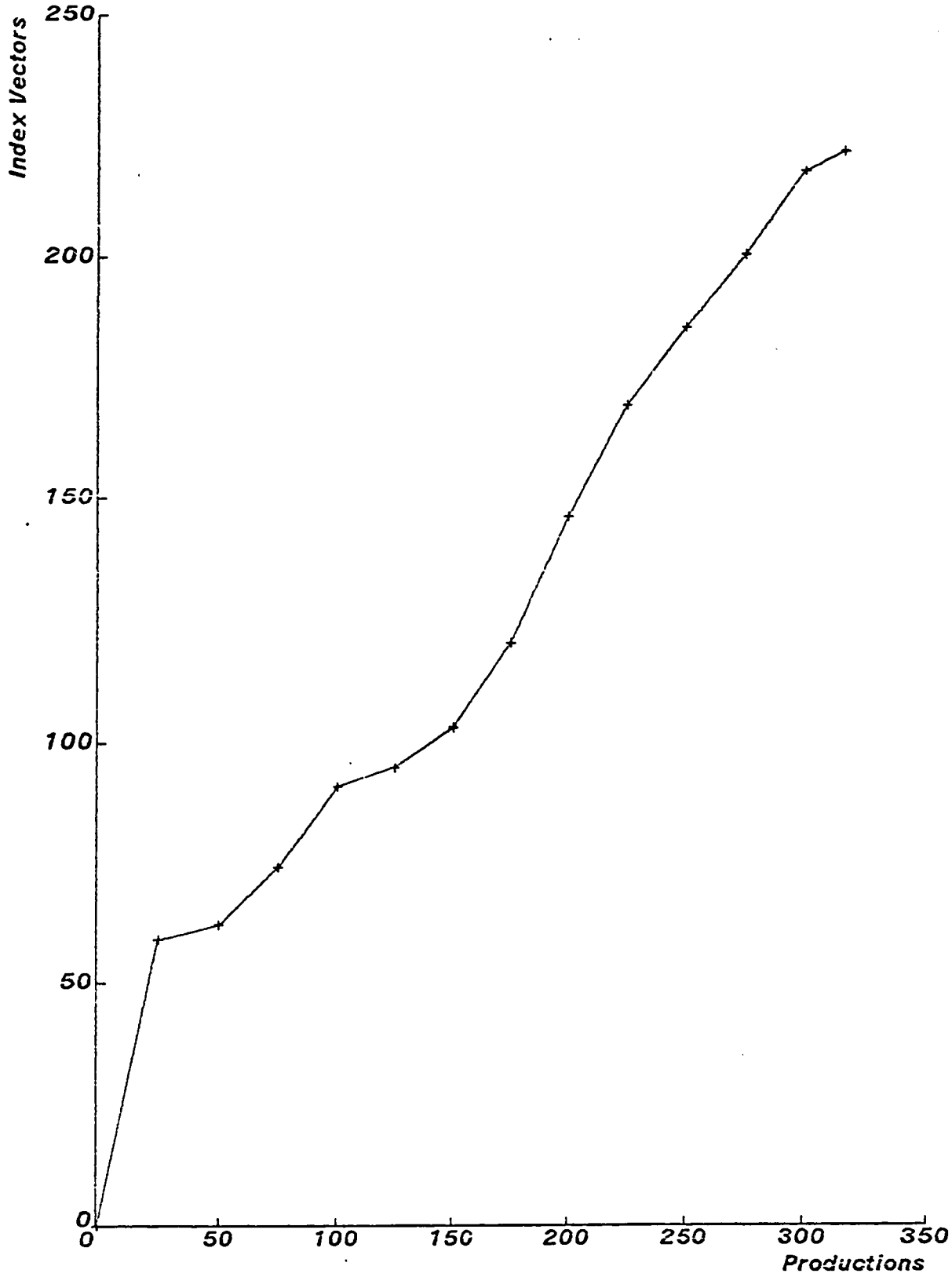
Graph 5.3. Rate of Network Growth: PH-632



Graph 5.4. Rate of Index Vector Growth: KERNL1



Graph 5.5. Rate of Index Vector Growth: HAUNT



Graph 5.6. Rate of Index Vector Growth: PH-632

single task.

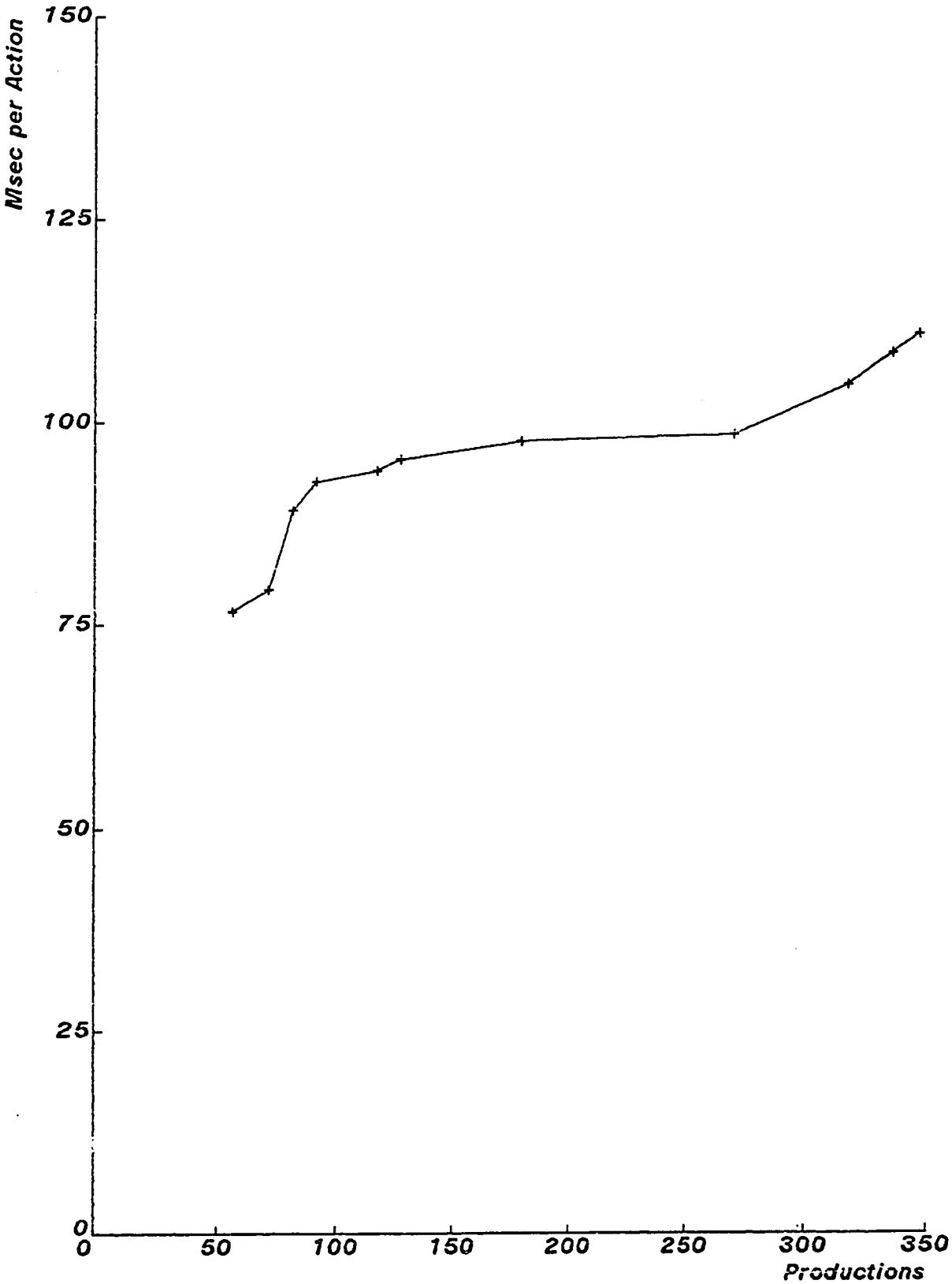
PH-632 was measured in a similar fashion. A task was chosen that could be performed by a group of 51 productions. The task was run six times, with production memory sizes ranging from 51 to 316 productions. Since no anomalies were found in the data for these runs, it was not necessary to use the finer divisions of the system that were used in measuring KERNL1. Working memory size constantly increases during the execution of PH-632. To minimize the effects of varying working memory size, three short runs were made, and working memory was reinitialized before every run. The three runs included a total of 104 productions firings and 361 actions. The average working memory size was 51 elements.

The experiments for HAUNT were executed in a more arbitrary fashion, resulting in somewhat less informative results. A typical run of the production system was monitored to determine which productions fired, and these productions were separated from the rest. There were 162 productions in this group. The remaining productions were arbitrarily divided into 6 groups of approximately equal size. The typical run was then repeated 7 times, with production memories ranging from 162 to 1017 productions. On each run 386 productions fired, performing 827 actions. Working memory averaged 75 elements. The problem with this experiment is that growth in the production system occured in an atypical manner; every run involved adding some productions that were sensitive to the same goals as the existing productions.

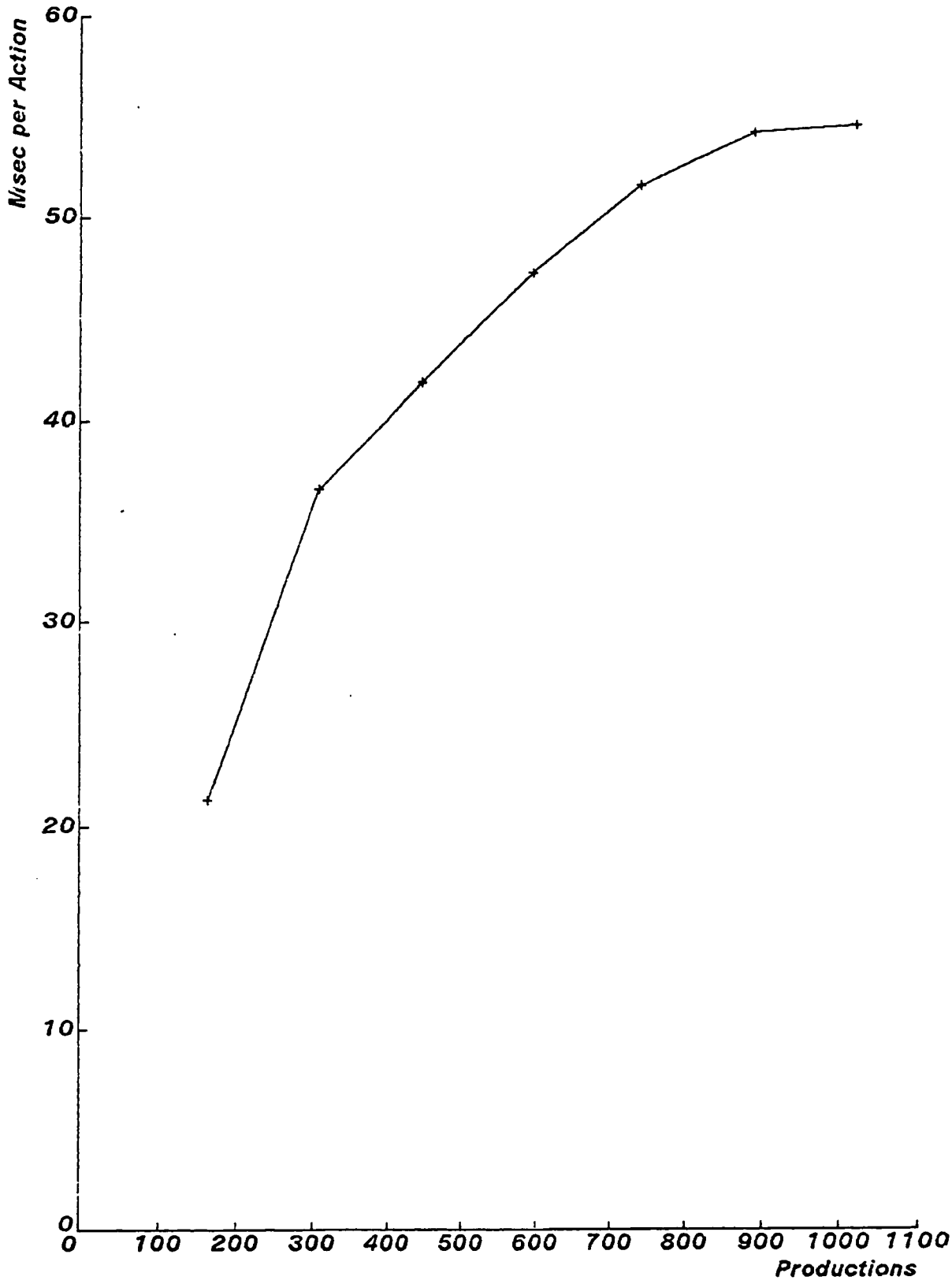
Graphs 5.7 through 5.9 show the effects of production memory size on time. The reason these graphs are irregular is that different parts of the match dominate the costs of each production system. To understand the effects of growth it is necessary to consider the number of node activations and the number of tests performed by the nodes.

Graph 5.10 shows the average number of nodes activated after each working memory change during the run of KERNL1; graph 5.11 shows the average number of tests performed.[1] These curves generally agree with the predictions made in Chapter 4. The number of one-input nodes activated increases almost linearly (though the effect of variation among individual productions is apparent also). The number of tests performed by two-input nodes remains constant. Over most of the measurements, the number of two-input node activations either remains constant or grows slowly. There are also discrepancies with the predictions, however. At one point (near 75 productions) a sharp increase occurs in the number of tests performed by the memory nodes and the number of activations of two-input and memory

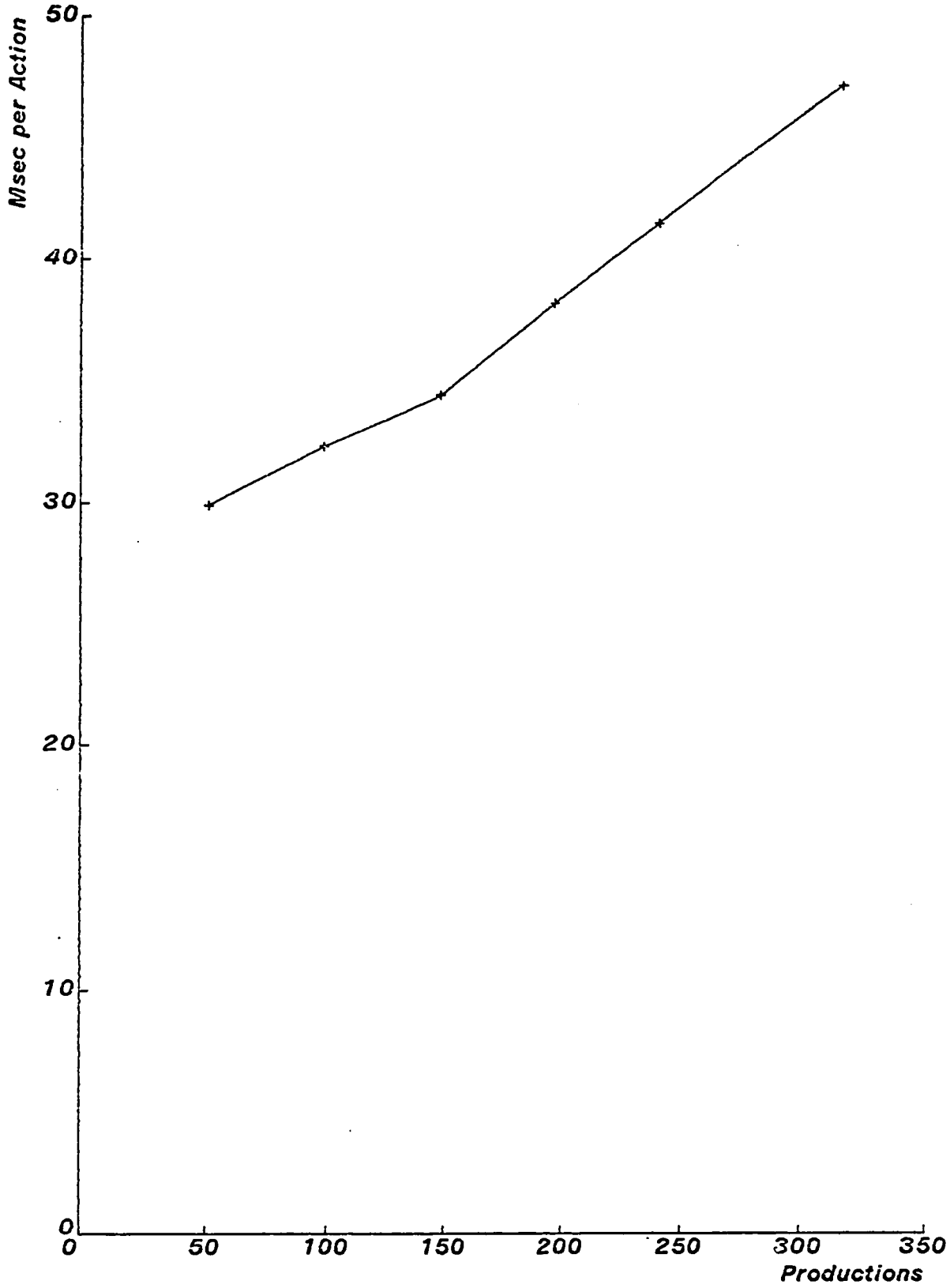
[1]The &P nodes have been left off the graphs for the three production systems because the results would have been unreadable on the scale used for the other nodes. The contribution of the &P nodes can be seen in tables 5.6 through 5.8.



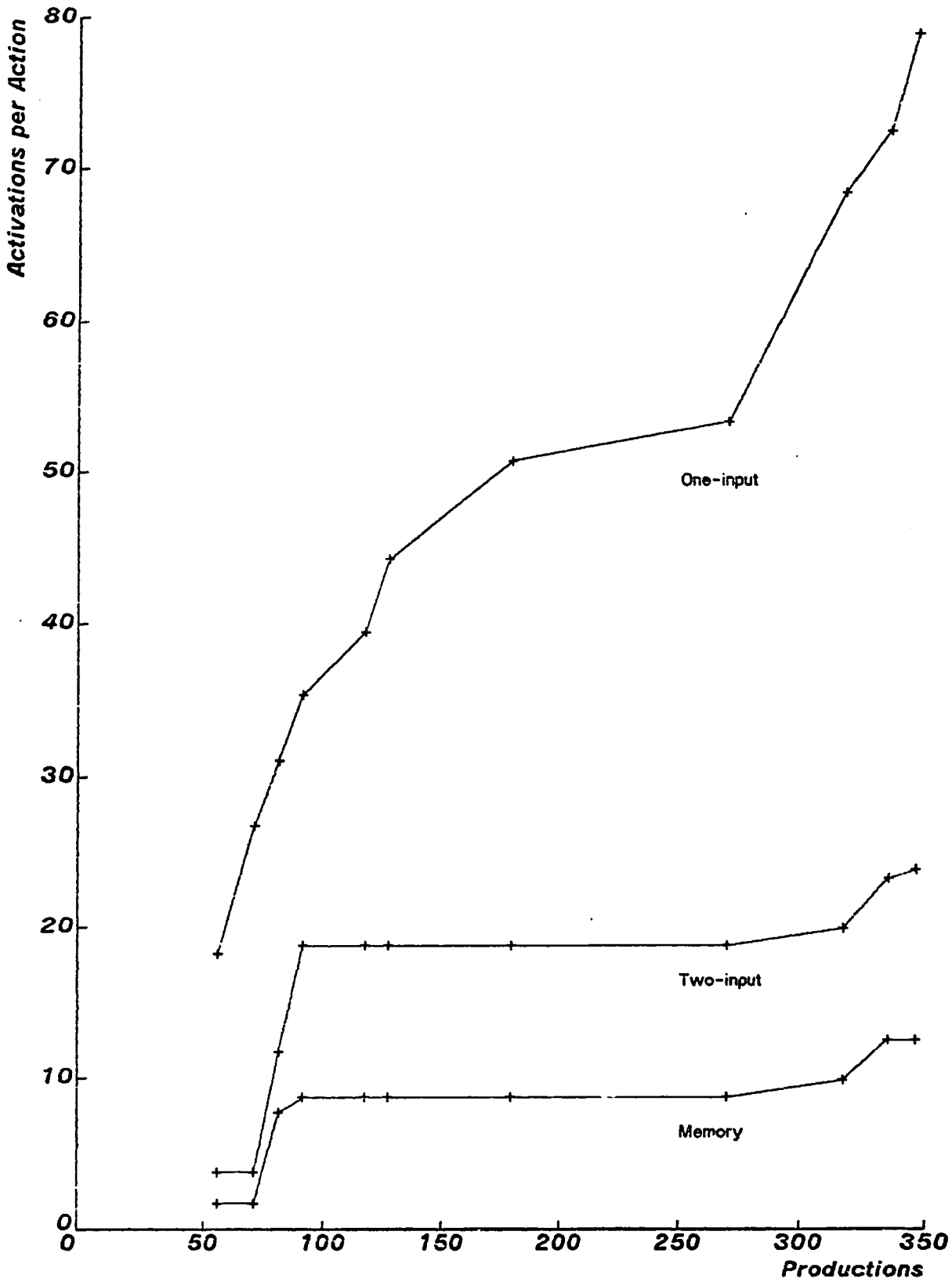
Graph 5.7. Effect of PM Size on Time: KERNL1



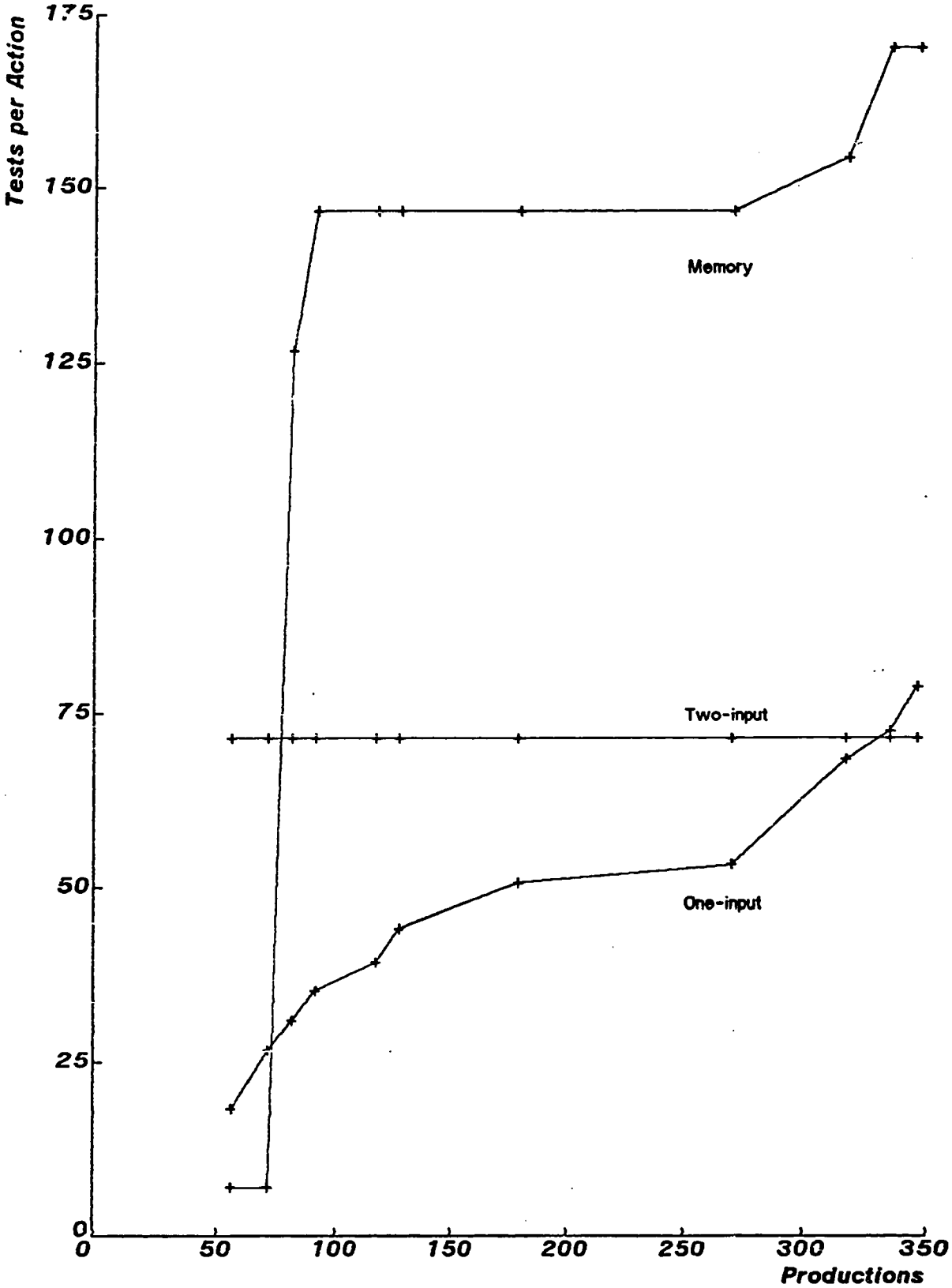
Graph 5.8. Effect of PM Size on Time: HAUNT



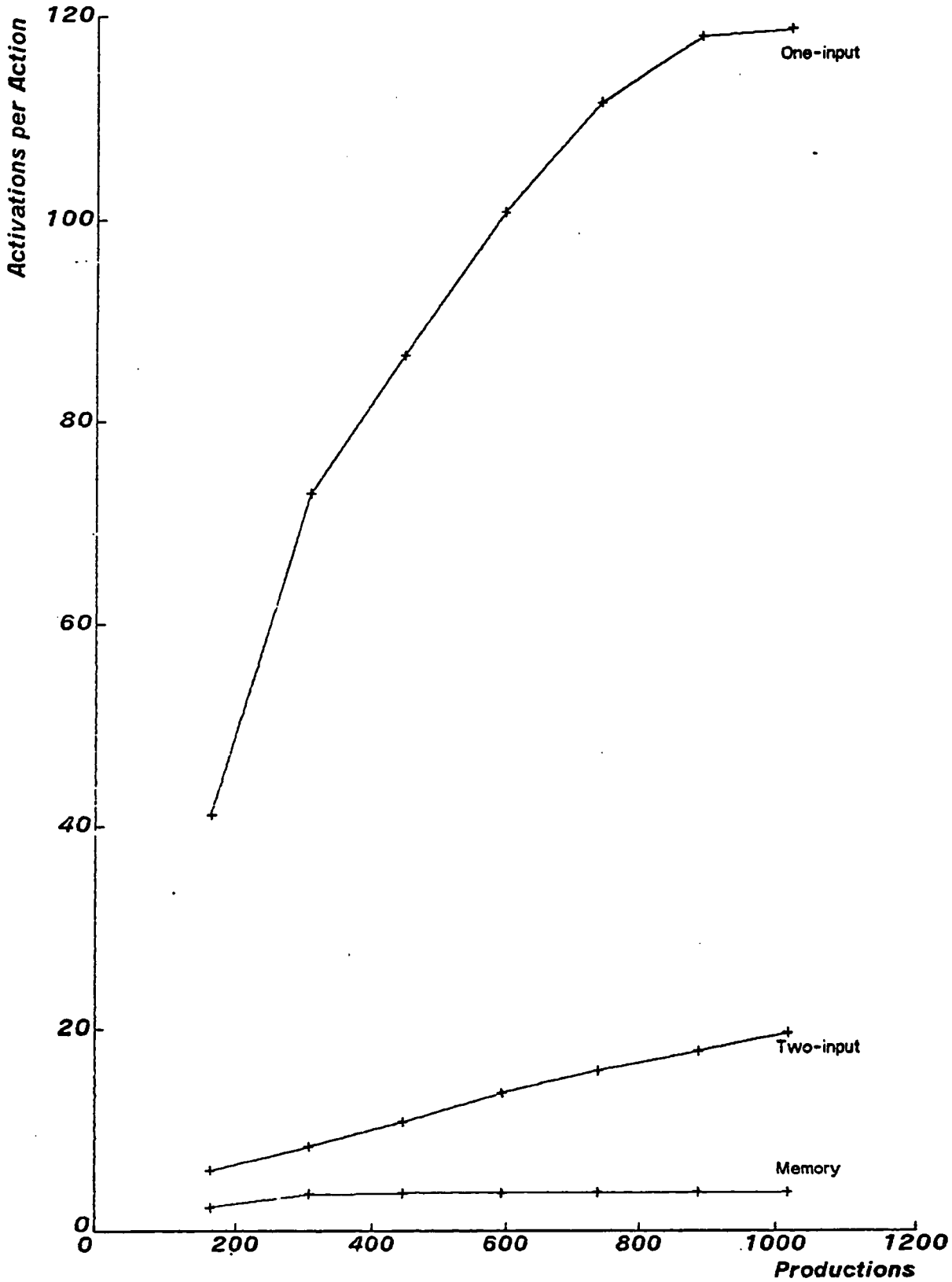
Graph 5.9. Effect of PM Size on Time: PH-632



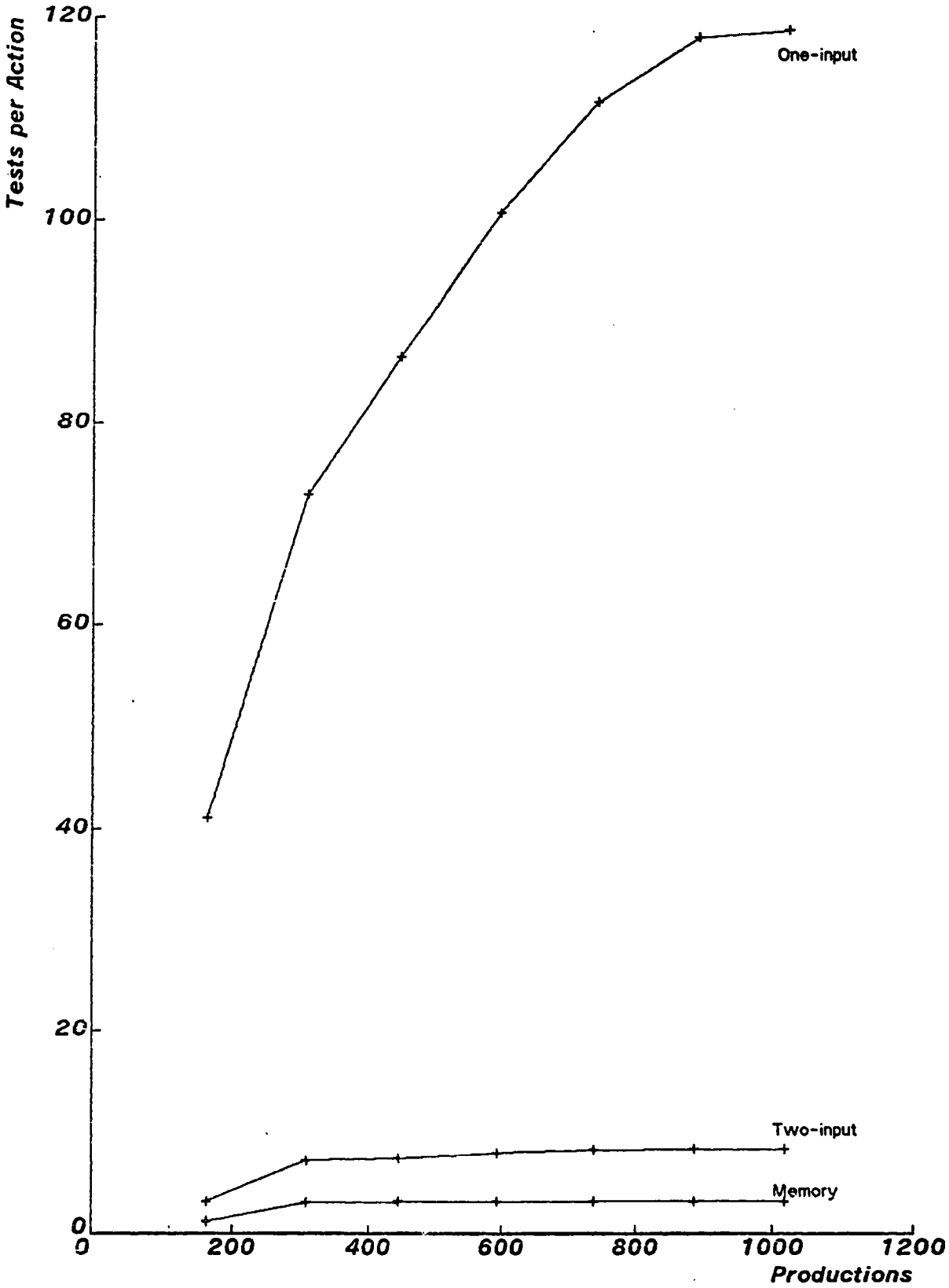
Graph 5.10. Effect of PM Size on Node Activations: KERNL1



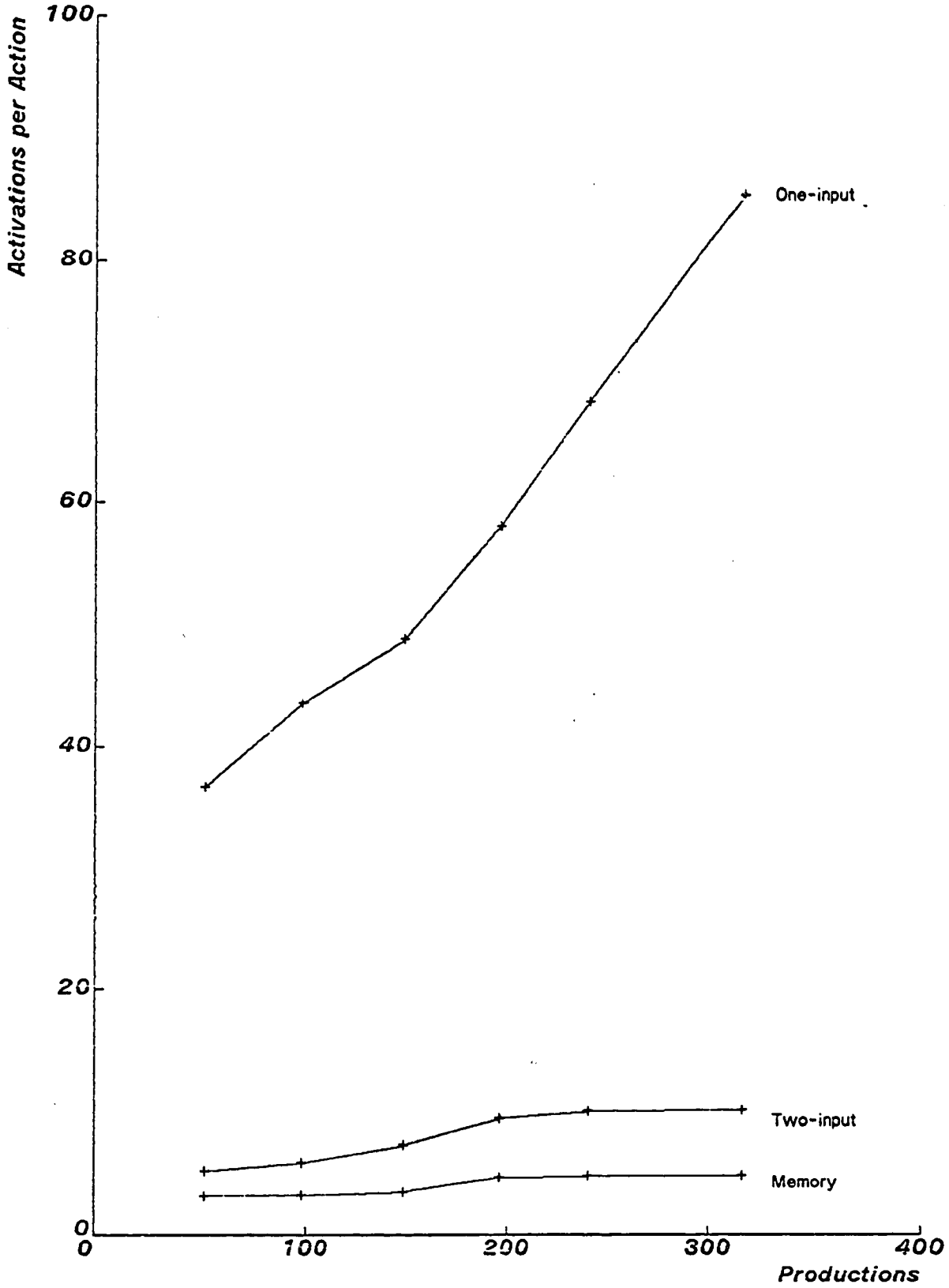
Graph 5.11. Effect of PM Size on Tests: KERNL1



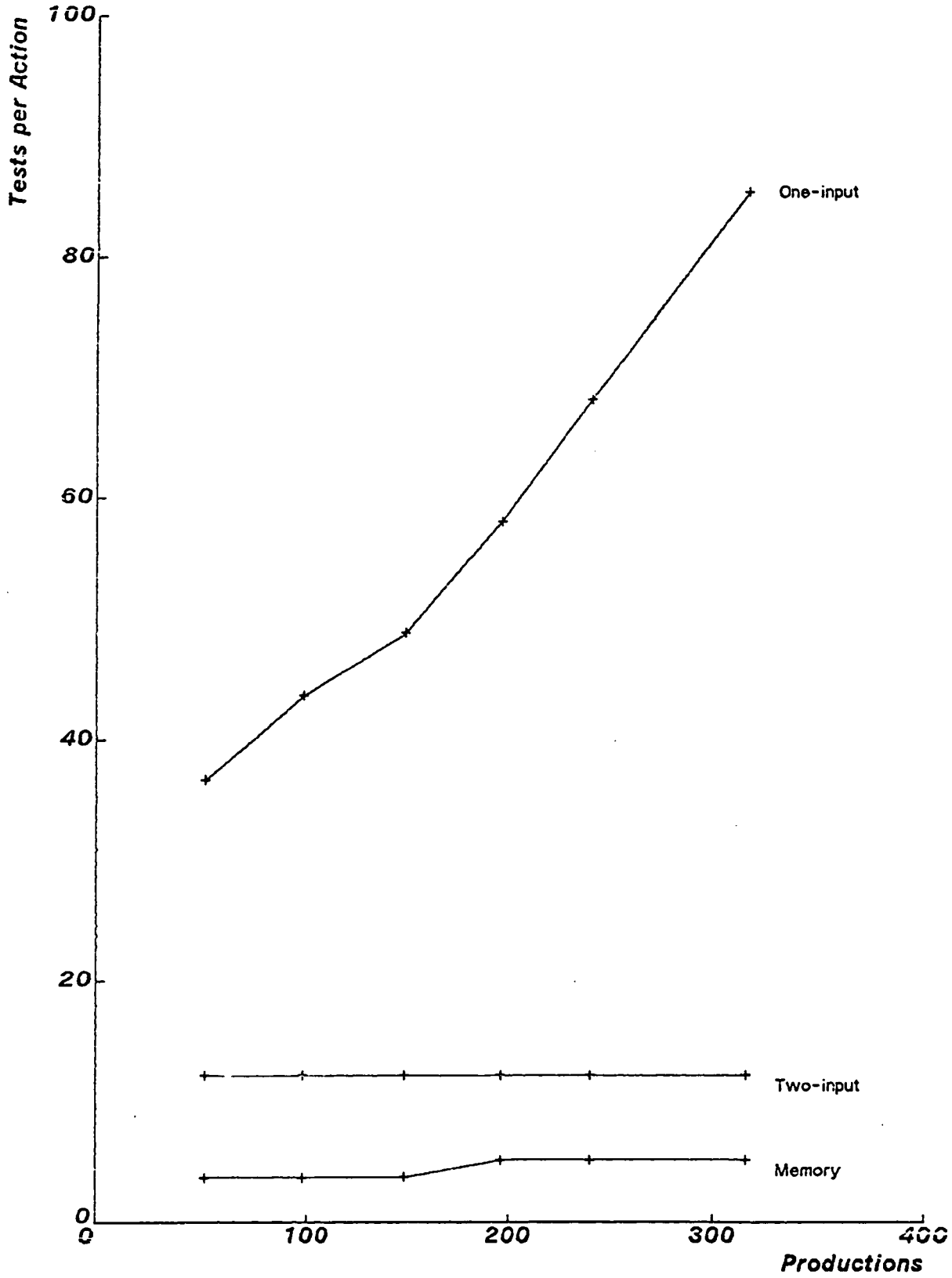
Graph 5.12. Effect of PM Size on Node Activations: HAUNT



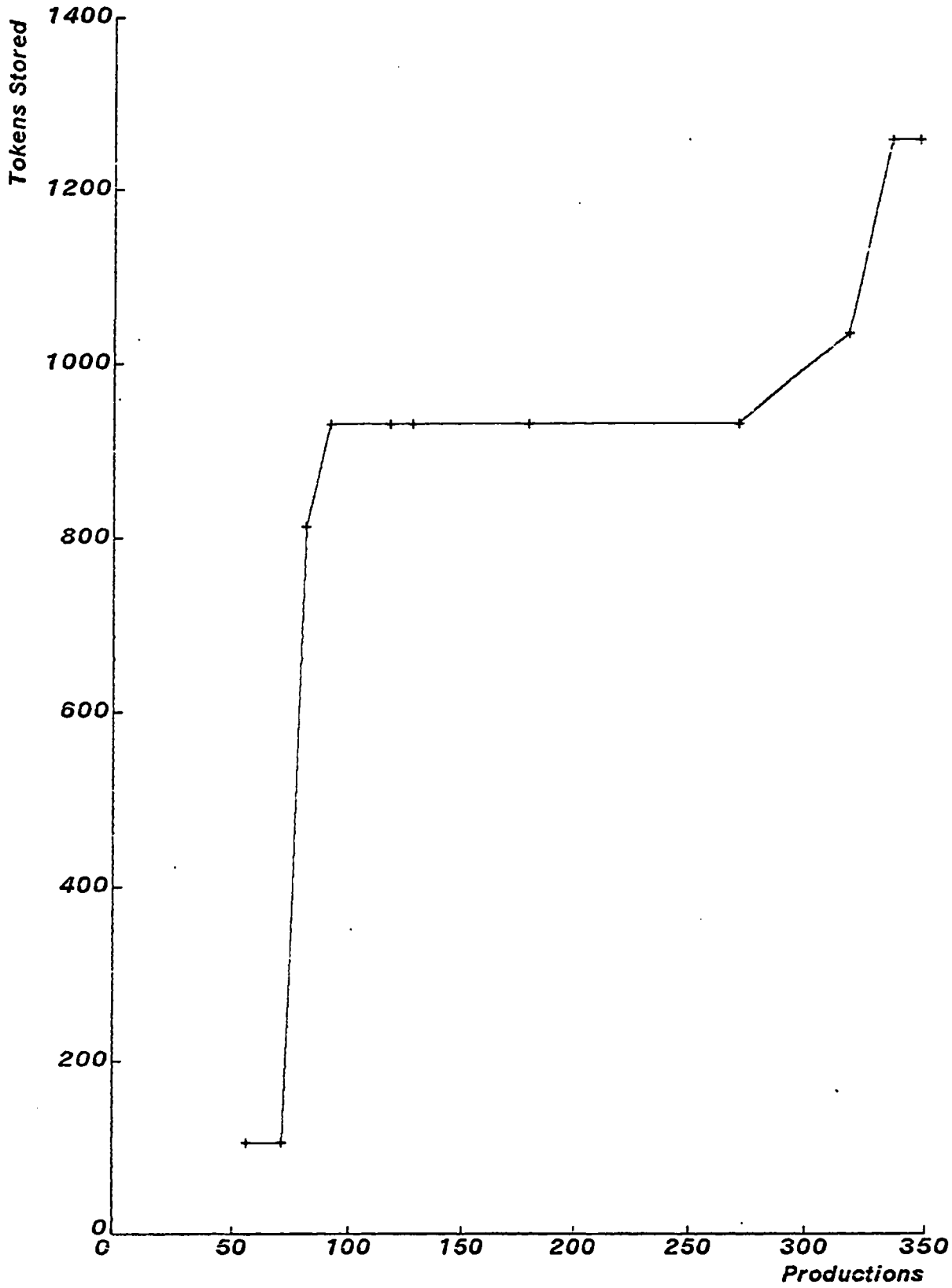
Graph 5.13. Effect of PM Size on Tests: HAUNT



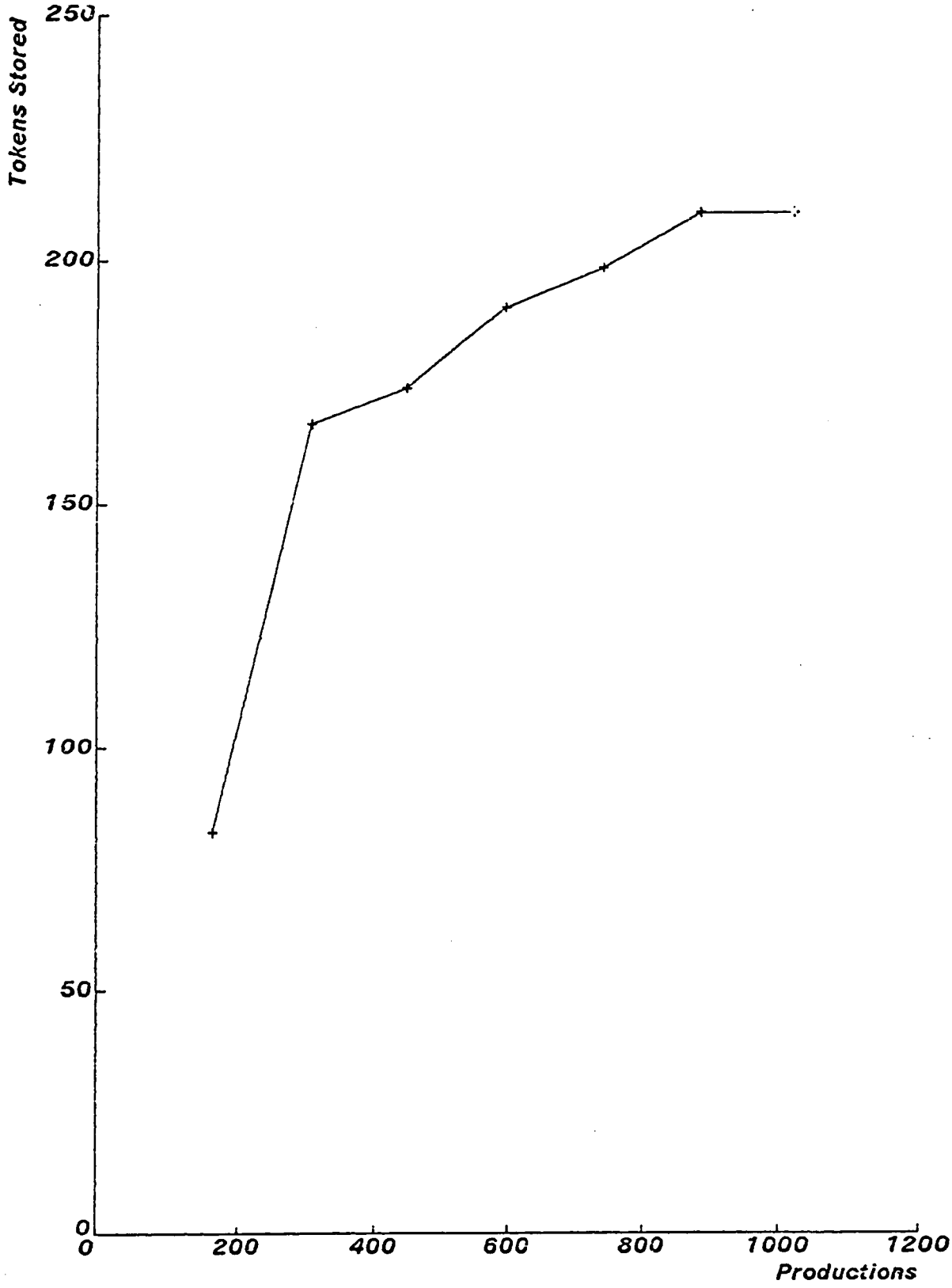
Graph 5.14. Effect of PM Size on Node Activations: PH-632



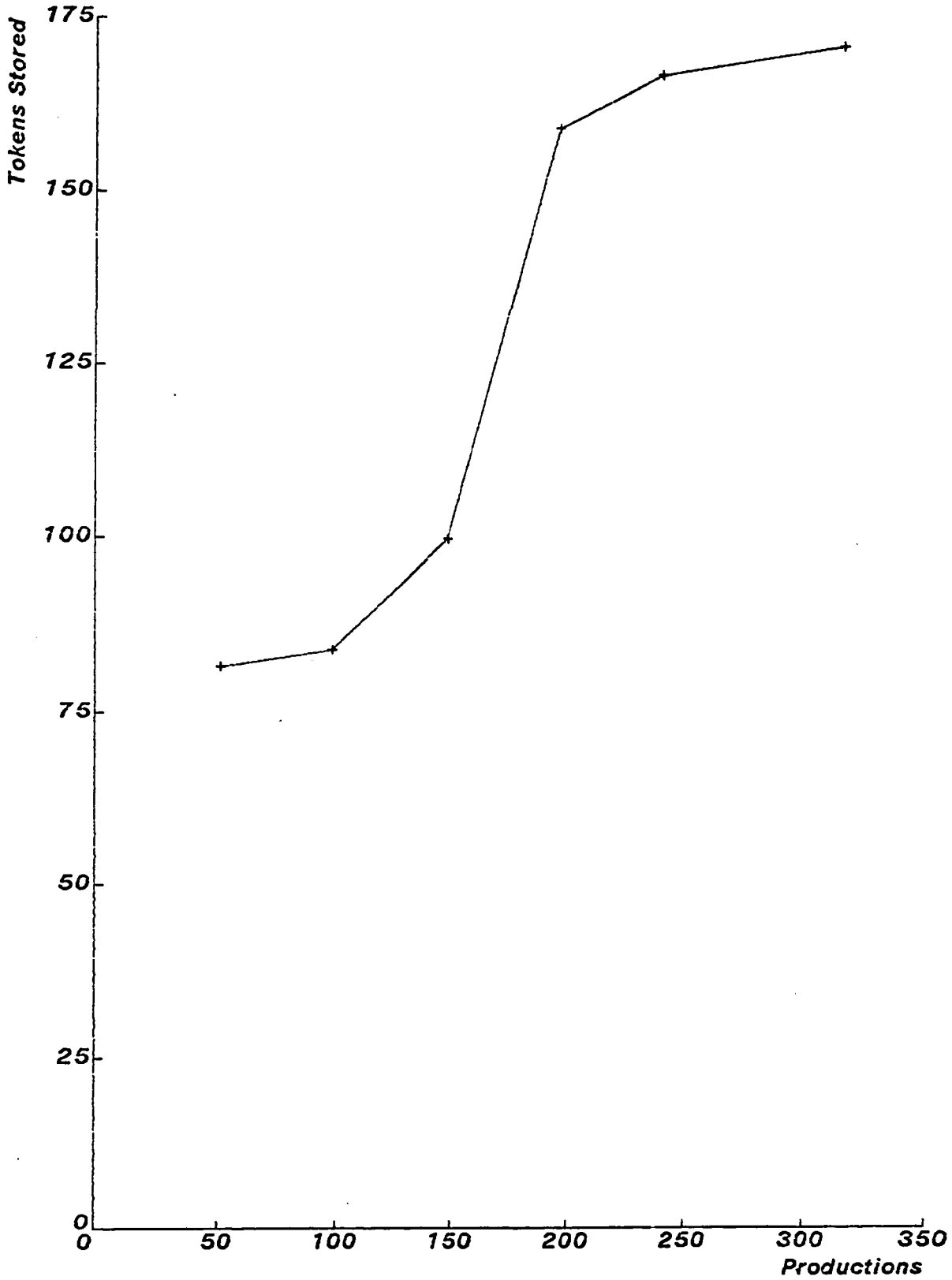
Graph 5.15. Effect of PM Size on Tests: PH-632



Graph 5.16. Effect of PM Size on Network Memories: KERNL1



Graph 5.17. Effect of PM Size on Network Memories: HAUNT



Graph 5.18. Effect of PM Size on Network Memories: PH-632

nodes. This can be explained by examining the productions read in. The productions give the system the ability to retry goals that have failed. Most of these "retry" productions have two condition elements. The first matches a goal requesting that another goal be retried, and the second matches that other goal. Production M12-2a is typical.

```
M12-2a  ( (retry unit  n =p) & =c1
          (=p =s =m !     & =c2 & #c1
          -->
          (tell phrase (W A) Being Retried ! =c2)
          =c2  (<delete> =c1) )
```

Since a large fraction of the data elements entering and leaving KERNL1's working memory are goals, the memory and two-input nodes for the condition elements like the second one in M12-2a are very active.

Graphs 5.12 and 5.13 show the average number of nodes activated and tests performed in interpreting HAUNT. Once again the predictions of the last chapter were found generally correct: there was little or no growth in the number of memory node activations, the number of tests performed by memory nodes, and the number of tests performed by two-input nodes. That the number of two-input node activations grew steadily can be explained by the atypical way productions were added to the system. Recall that each run after the first involved adding productions that matched some of the data elements used in the run. The apparent sublinear growth in the number of one-input node activations is also probably an effect of the atypical adding of productions.

Graphs 5.14 and 5.15 show the effects of growth in PH-632. This time the predictions of Chapter 4 seem to be exactly right. The number of one-input node activations grows approximately linearly. The number of activations of memory and two-input nodes grow slowly. The tests performed by memory and two-input nodes remain almost constant in number.

Graphs 5.16 through 5.18 show the effects of production memory size on the number of tokens stored in the node memories. The curves are irregular because the variations among the productions dominate the results. The increase caused by KERNL1's "retry" productions, for example, is more than half the total increase.

5.4.3 The Cost Formula

The following tables compare the observed costs of executing the production systems to the costs predicted by the formula from section 5.1.6. The computed cost includes the cost

of one evaluation of a RHS function (E) plus the amount of per cycle overhead that should be charged to each action (O/A) plus the total time costs of the various parts of the match listed below. The second and third columns show the measured and computed time cost of processing one working memory change. The remaining columns break the computed cost down, showing the cost of each class of operation. All times are in milliseconds. The parenthesized numbers are the counts of the number of times each operation was performed. These numbers repeat the information in graphs 5.10 through 5.15.

Prods.	Meas.	Comp.	N1	N2	T2	Nm	Tm	Np
56	76.7	76.1	6.41 (18.3)	0.49 (3.78)	59.3 (71.5)	0.22 (1.73)	0.40 (6.94)	2.00 (1.00)
71	79.3	79.1	9.38 (26.8)	0.49 (3.78)	59.3 (71.5)	0.22 (1.73)	0.40 (6.94)	2.00 (1.00)
81	89.2	89.4	10.9 (31.1)	1.53 (11.8)	59.3 (71.5)	1.00 (7.73)	7.37 (127)	2.00 (1.00)
91	92.7	93.1	12.4 (35.4)	2.44 (18.8)	59.3 (71.5)	1.13 (8.73)	8.53 (147)	2.00 (1.00)
117	94.1	94.5	13.8 (39.5)	2.44 (18.8)	59.3 (71.5)	1.13 (8.73)	8.53 (147)	2.00 (1.00)
127	95.4	96.2	15.5 (44.3)	2.44 (18.8)	59.3 (71.5)	1.13 (8.73)	8.53 (147)	2.00 (1.00)
179	97.6	98.5	17.8 (50.8)	2.44 (18.8)	59.3 (71.5)	1.13 (8.73)	8.53 (147)	2.00 (1.00)
270	98.4	99.4	18.7 (53.4)	2.44 (18.8)	59.3 (71.5)	1.13 (8.73)	8.53 (147)	2.00 (1.00)
317	105	105	23.9 (68.4)	2.59 (19.9)	59.3 (71.5)	1.28 (9.84)	8.93 (154)	2.06 (1.03)
335	108	108	25.3 (72.4)	3.02 (23.2)	59.3 (71.5)	1.63 (12.5)	9.86 (170)	2.06 (1.03)
346	111	111	27.6 (78.8)	3.09 (23.8)	59.3 (71.5)	1.63 (12.5)	9.86 (170)	2.06 (1.03)

Table 5.6. Cost Breakdown for KERNL1

Prods.	Meas.	Comp.	N1	N2	T2	Nm	Tm	Np
162	21.3	21.0	13.2 (41.2)	0.79 (6.05)	2.10 (3.14)	0.31 (2.36)	0.07 (1.19)	1.84 (1.67)
306	36.6	34.9	23.4 (73.0)	1.10 (8.47)	4.82 (7.19)	0.47 (3.65)	0.18 (3.05)	2.24 (2.04)
444	41.8	39.7	27.7 (86.5)	1.42 (10.9)	4.94 (7.38)	0.49 (3.78)	0.18 (3.11)	2.26 (2.05)
592	47.2	45.0	32.3 (101)	1.79 (13.8)	5.31 (7.92)	0.50 (3.81)	0.18 (3.13)	2.26 (2.05)
737	51.5	49.6	35.8 (112)	2.08 (16.0)	5.48 (8.18)	0.50 (3.82)	0.18 (3.13)	2.26 (2.05)
886	54.1	51.3	37.8 (118)	2.33 (17.9)	5.54 (8.27)	0.50 (3.85)	0.18 (3.14)	2.27 (2.06)
1017	54.5	51.8	38.1 (119)	2.55 (19.6)	5.54 (8.2.)	0.50 (3.85)	0.18 (3.14)	2.27 (2.06)

Table 5.7. Cost Breakdown for HAUNT

Prods.	Meas.	Comp.	N1	N2	T2	Nm	Tm	Np
51	29.9	28.8	12.0 (36.7)	0.68 (5.23)	10.9 (12.2)	0.42 (3.23)	0.20 (3.74)	1.72 (1.01)
98	32.3	31.1	14.2 (43.7)	0.77 (5.89)	10.9 (12.2)	0.43 (3.27)	0.20 (3.74)	1.72 (1.01)
148	34.4	32.5	15.9 (48.8)	0.95 (7.34)	10.9 (12.2)	0.46 (3.57)	0.22 (3.77)	1.72 (1.01)
196	38.1	36.5	18.9 (58.0)	1.23 (9.45)	10.9 (12.2)	0.61 (4.71)	0.30 (5.21)	1.73 (1.02)
240	41.4	40.0	22.3 (68.3)	1.30 (10.0)	10.9 (12.2)	0.63 (4.83)	0.30 (5.21)	1.73 (1.02)
316	47.1	45.6	27.8 (85.3)	1.33 (10.2)	10.9 (12.2)	0.63 (4.88)	0.30 (5.21)	1.73 (1.02)

Table 5.8. Cost Breakdown for PH-632

There is close agreement between the measured and computed times for KERNL1. There is less agreement between the measured and computed times for HAUNT and PH-632. But even in those systems there is no relation between size and error; the relative errors did not generally increase with increases in the size of production memory.[1] Thus it seems safe to claim that all the effects of production memory size have been captured in the measurements.

5.4.4 The Effects of WM Size on Working Space and Time

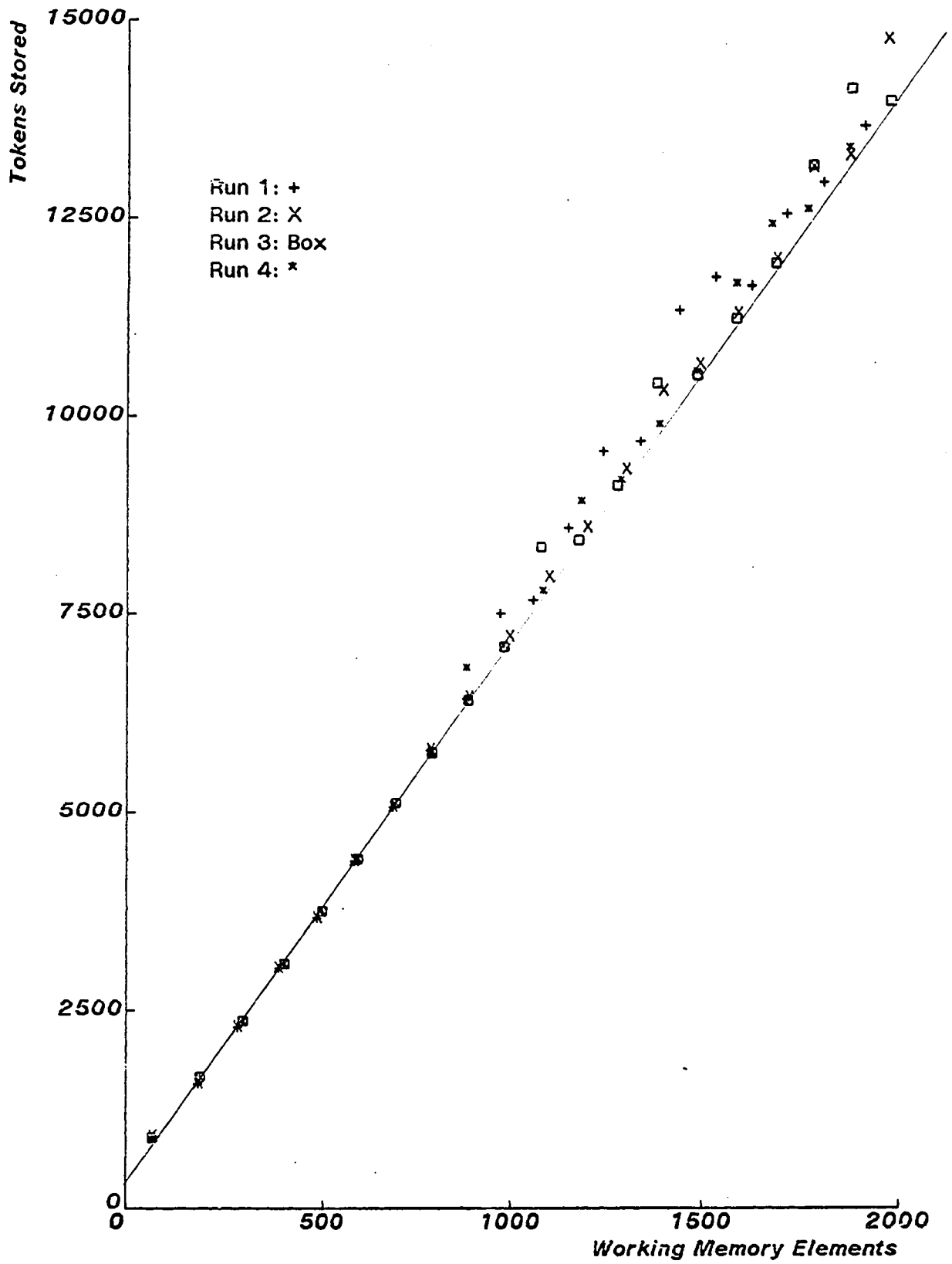
Only KERNL1 was measured to determine the effects of working memory size because it had two features making it uniquely suited to the experiments. First, it relied on the automatic deletion feature of OPS2 to rid its working memory of old data elements. With automatic deletion disabled, therefore, its working memory grew steadily. Second, the task it performed, understanding a subset of English, was complex enough to allow extremely long runs. The working memories of the other two production systems also grew steadily, but those systems could not easily be made to run for long periods.

The experiments involved 4 runs. About 10,000 actions were performed during each run, giving a final working memory size in each case of about 2000 elements. The interpreter was modified so that it would pause after each 500 actions to print out various statistics about those 500 actions. The task performed during the runs was accepting instructions and building productions to carry out the instructions. OPS2's production building mechanism was disabled for the experiment, however, so the size of production memory remained constant at 261 productions. The disabling did not affect the processing of the system because it was carried out in a manner that could not be detected by the productions.

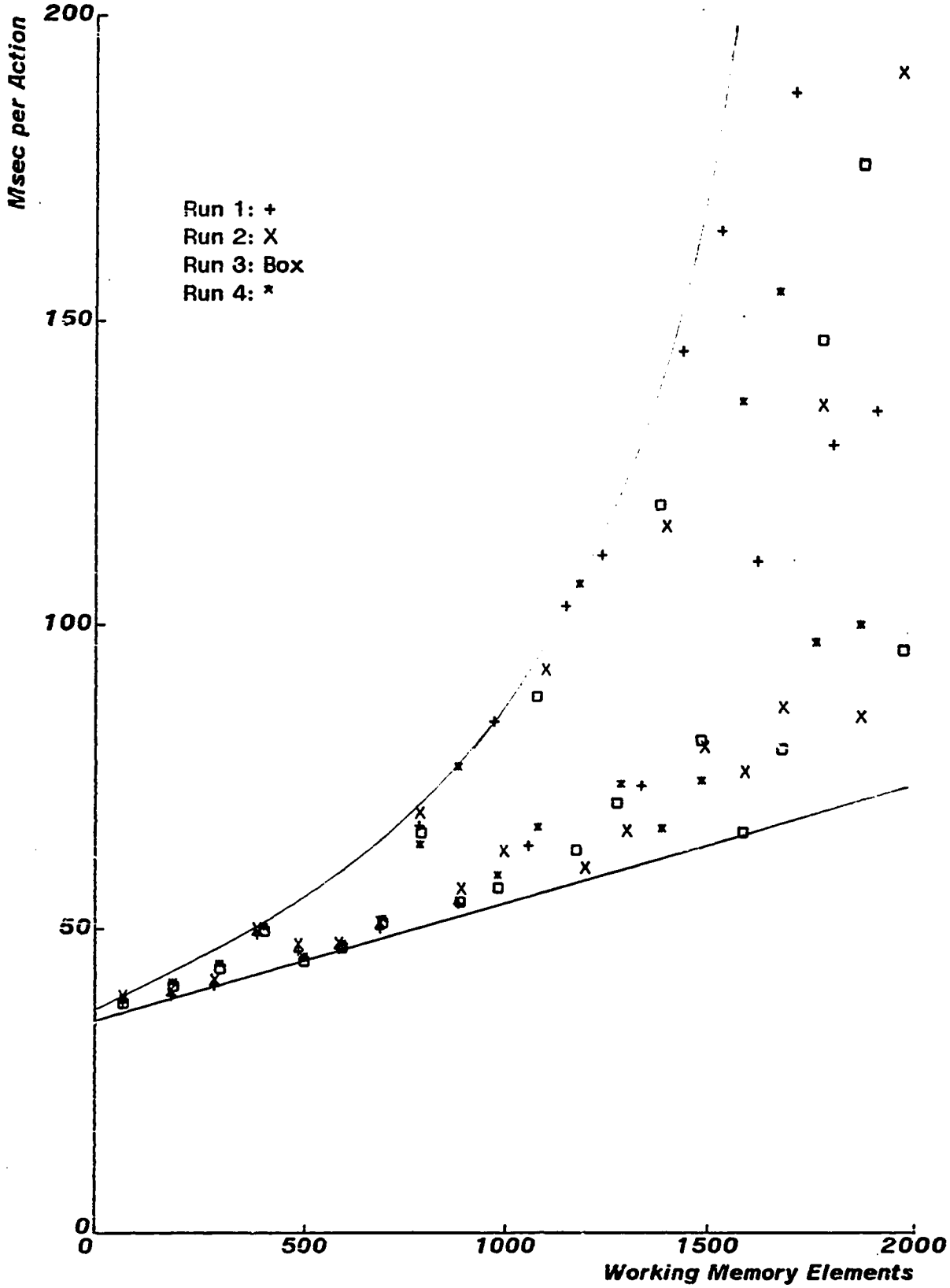
Graphs 5.19 and 5.20 summarize the results of the experiment. The number of tokens stored generally increased linearly with the number of elements in working memory. The time to process one action varied greatly. The measured values fill the space between a linear and a higher order polynomial. The reason for the variation was that different productions were being instantiated during the measured intervals. Graphs 5.21 and 5.22 show the effects in Run 1 in more detail. (Only one run is shown because the other three were almost identical.) The number of node activations of all kinds remained almost constant. The number of tests performed by two-input nodes increased at a linear rate. The number of tests performed by the memory nodes oscillated wildly. These tests were the cause of the variations in overall time costs.

---

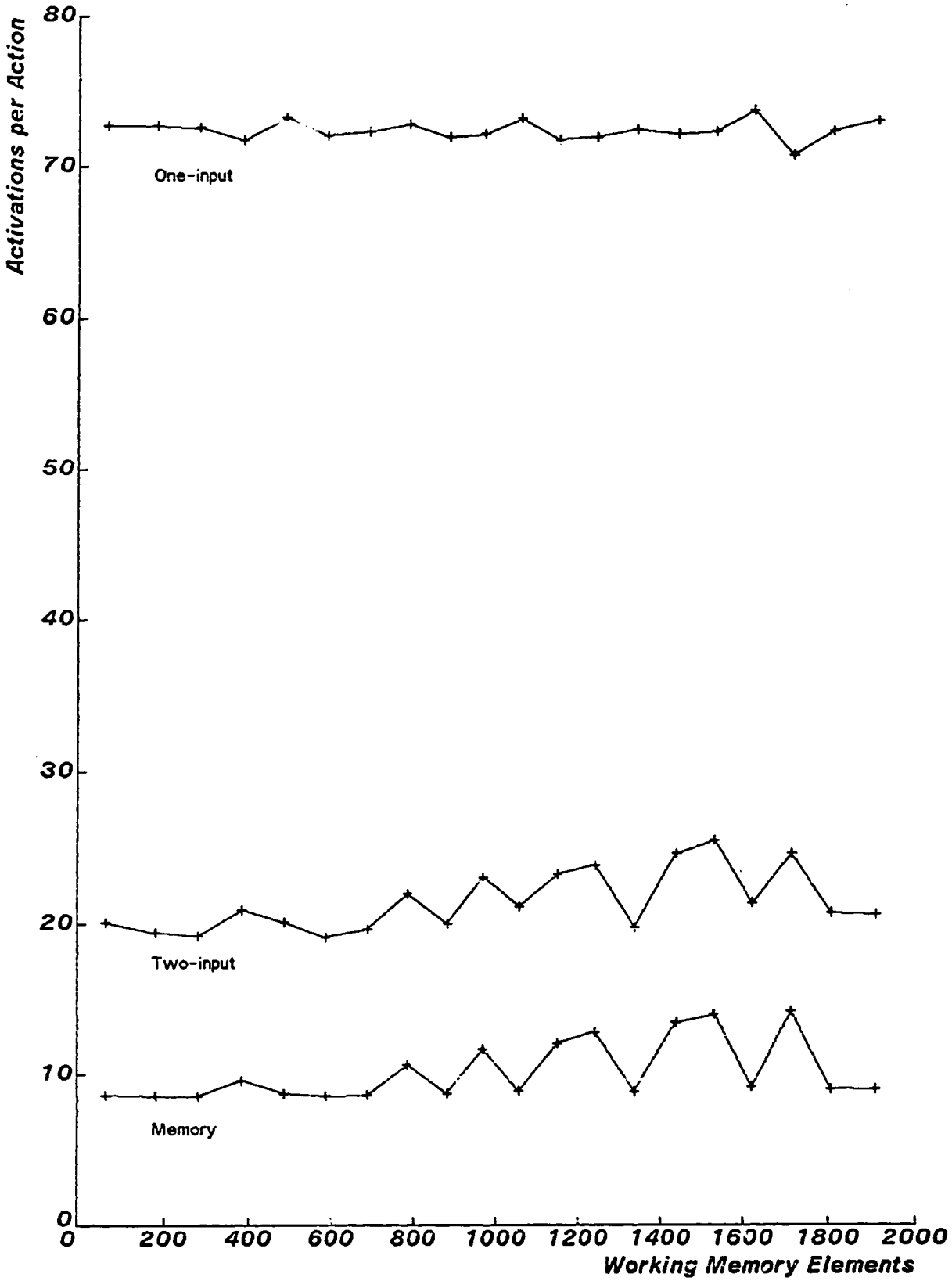
[1] The relative errors in HAUNT are (in order, from the smallest system to the largest) 1.4%, 4.6%, 5.0%, 4.7%, 3.7%, 5.2%, and 5.0%. The relative errors for PH-632 are 3.7%, 3.7%, 5.5%, 4.2%, 3.4%, and 3.2%.



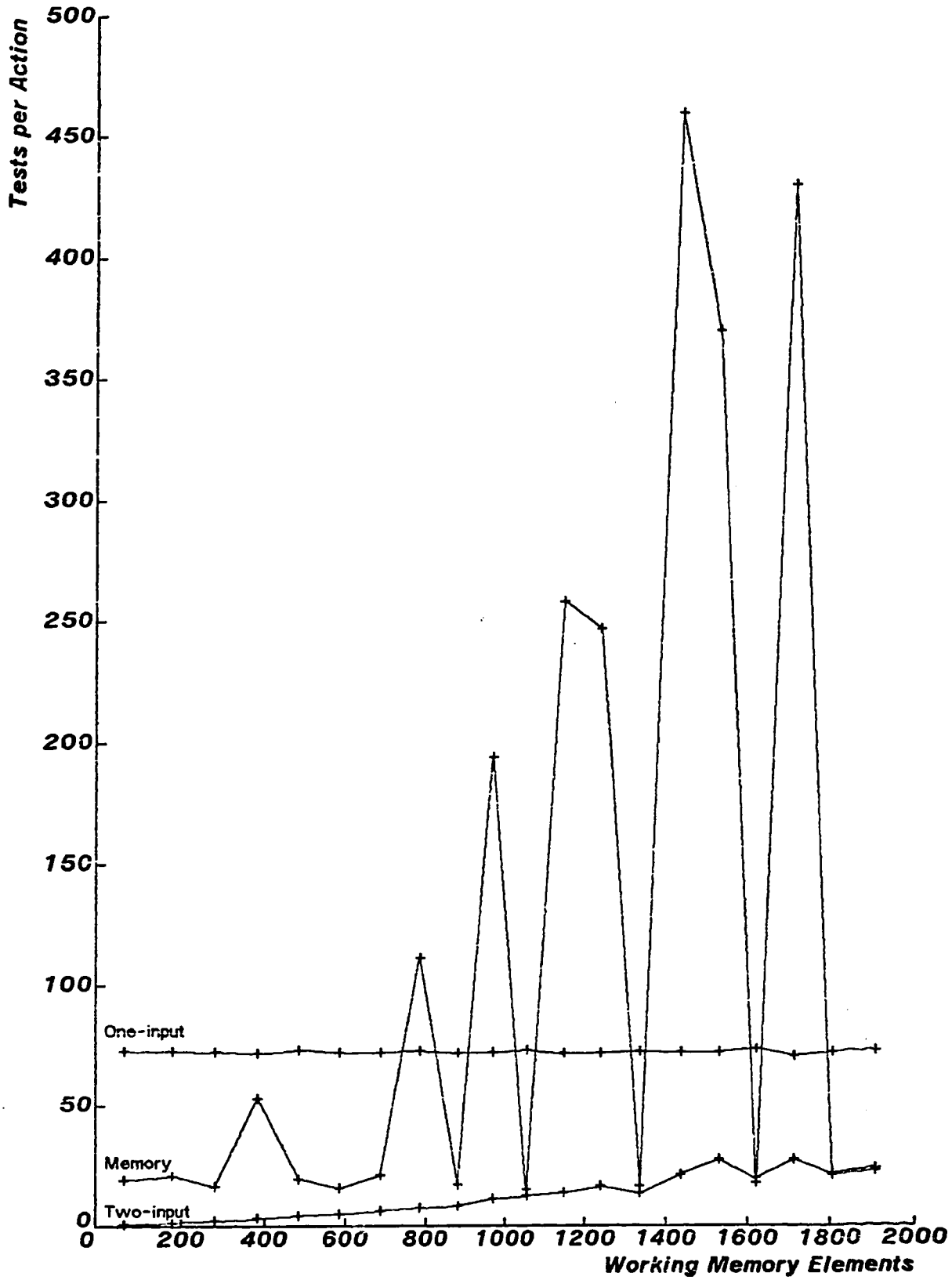
Graph 5.19. Effect of WM Size on Network Memories



Graph 5.20. Effect of WM Size on Time



Graph 5.21. Effect of WM Size on Node Activations: Run 1



Graph 5.22. Effect of WM Size on Tests: Run 1

5.5 Testing the Assumptions in Chapter 4

If the arguments made in Chapter 4 about the ubiquity of renaming are correct, there should be a noticeable effect on the network. The network should contain a few nodes with unusually long successor fields, and the nodes pointed to by the links in the fields should be &ATOM nodes. To test the arguments, then, one can simply examine the successor fields in the network. The following table shows the frequency of occurrence of fields of various lengths.

Length	In KERNL1	In HAUNT	In PH-632
0	407	914	872
1	2145	2920	2248
2	204	373	265
3	56	72	71
4	16	48	32
5	10	25	15
6	5	15	7
7	2	4	12
8	1	14	6
9	1	3	6
10	1	4	3
11	1	7	1
12	2	3	1
13	1	2	1
14	0	0	1
15	2	0	2
16	1	2	2
17	2	1	1
18	0	2	2
19	0	1	0
20	0	3	1
21	0	2	0
24	0	0	1
25	0	1	0
26	0	0	2
29	0	1	0
31	0	1	0
34	0	2	0
39	0	1	0
40	0	1	0
41	0	1	0
42	0	1	0
43	0	1	0
46	1	0	0
48	1	1	1
51	0	1	0
55	0	1	0
69	0	2	0

Table 5.9. Successor Field Lengths

The KERNL1 network contains 2 successor fields that are unusually long (more than twice the length of the next longest field). The one with 46 elements contains only links to &ATOM nodes, as predicted. The one with 48 elements contains links to &VEX nodes. The distribution of field lengths in the HAUNT network contains no gaps like the one from 17 to

46 in the KERNL1 network. Taking 50 as an arbitrary cutoff, one has four fields to examine, two with 69 elements, one with 55, and one with 51. The two longest fields both hold pointers to &ATOM nodes. The field with 55 elements holds 54 pointers to &ATOM nodes and one pointer to an &MEM node. The field with 51 elements holds pointers to &VEX nodes. The PH-632 network contains only one unusually long field. This field contains 47 elements, 46 of which are links to &ATOM nodes. The remaining element is a link to an &LEN node.

These results are not inconsistent with the conclusions reached in Chapter 4. Obviously they provide some support for the contention that networks will contain nodes that have many &ATOM nodes for immediate successors. The presence of the &VEX pointers can be explained by the size of the production systems. The discussion of growth in the last chapter predicted that productions for different goals would sometimes be sensitive to the same data elements, but that the number of productions sensitive to a given data element would be a sublinear function of the number of productions in the system. If the argument there was correct, finding lists of pointers to &VEX nodes as long as the lists of pointers to &ATOM nodes is a result of having measured such small production systems. If the systems had been ten times larger, the longest successor fields would probably all have contained pointers to &ATOM nodes.

5.6 Using Hardware Efficiently

Two realities of hardware technology impact strongly on current research into computer architecture: (1) parallel machines are less expensive than serial machines of comparable speed, and (2) the primary memories of computers often have appreciably longer cycle times than the processors. The first of these has caused many parallel computers to be designed. The second has caused the designers to explore caching (that is, using automatic techniques to migrate information between primary memory and a small, fast memory called the cache) and fetching more than one word on each memory access. Caching gives improved performance because most programs show some locality in their references to memory; if a given location is accessed, there is a high probability that a nearby location will be accessed soon. Fetching several words at once improves performance because there is an especially high likelihood that word K+1 will be needed soon after word K is requested.

The experiments in this section were performed in order to determine how much potential for parallel execution exists in the OPS2 interpreter and to determine whether fetching several nodes at once would speed up the execution significantly. No experiments were performed to determine the effects of caching because the production systems were too small. (If a small production system shows a potential for parallel execution, then it is safe to assume that larger systems will show at least as much potential; one cannot make a similar

assumption concerning use of caching because increases in program size make caching less efficacious.)

5.6.1 Parallel Execution

The potential for parallel execution was determined simply by counting the average number of nodes awaiting execution. Each time the interpreter selected a node, a count was made of the number of nodes from which it could choose. These counts were summed and divided by the number of times the selection was made, giving an estimate of the average degree of parallel activity that is possible. The experiments were performed twice for each production system, once with the smallest production system used in the production memory size experiments, and once with the largest. The following table contains the results. The average number of nodes activated on each cycle is also given so that a comparison can be made between the total amount of work performed and the amount that could have been performed in parallel. The ratio of these two is low in all cases, suggesting that large production systems could make good use of parallelism.

	Nodes Pending	Nodes Activated	Ratio
KERNL1 (56 Ps)	6	24	4
KERNL1 (346 Ps)	15	115	7.7
HAUNT (162 Ps)	9	50	5.6
HAUNT (1017 Ps)	26	145	5.6
PH-632 (51 Ps)	14	45	3.2
PH-632 (316 Ps)	19	100	5.3

Table 5.10. Potential Parallelism

5.6.2 Using Wider Memories

If two nodes are to be fetched when one is requested, the obvious extra node to fetch is the immediate descendant of the requested node; if the test at the node succeeds, the descendant will be the next node activated. Since the descendant node will not be needed if the test at the node fails, whether this will provide any significant speed increase depends on the fraction of the tests that fail. The following table contains the probabilities of success at the one-input nodes. As can be seen, the probability is low, and it decreases when larger production memories are used. Fetching the immediate successors of nodes would, it seems, give little increase in speed.

	Successes
KERNL1 (56 Ps)	38.9%
KERNL1 (346 Ps)	31.4%
HAUNT (162 Ps)	17.2%
HAUNT (1017 Ps)	6.6%
PH-632 (51 Ps)	23.0%
PH-632 (316 Ps)	13.2%

Table 5.11. Successful Tests at One-input Nodes

Another possibility for the use of wider memories is to store nodes adjacent to their brother nodes. The interpreter could be modified so that after processing a given node, it would next process one of that node's brothers. Then when a node is fetched, one or more of the node's brothers could be fetched as well. If the node has any brothers, they will necessarily be activated, so this strategy eliminates the problem of fetching nodes unnecessarily. A set of runs was made to determine the likelihood that an activated node will have a brother to be fetched along with it. The results are shown in the following table. Since the likelihood increases with increases in the size of the production system, using wider

memories and prefetching brothers would probably be quite effective with large production systems.

	Brothers
KERNL1 (56 Ps)	51.0%
KERNL1 (346 Ps)	63.8%
HAUNT (162 Ps)	73.5%
HAUNT (1017 Ps)	88.7%
PH-632 (51 Ps)	67.1%
PH-632 (316 Ps)	80.7%

Table 5.12. Activated Nodes Having Brothers

5.7 Changes to the OPS2 Algorithm

This chapter, like the last, has uncovered a weakness in the OPS2 algorithm. The experiments described in section 5.4.4 showed that the time required to update the node memories is very sensitive to the size of working memory. The reason for the sensitivity is that the node memories are organized as linear lists; to alleviate it, it is necessary only to reorganize the memories -- as hash tables, perhaps. One might not want to make every memory in the network a hash table since that would increase the size of the network. Instead one might want the interpreter to convert the linear lists into hash tables automatically when the lists become excessively long.

6. A Machine Architecture for Production Systems

This chapter deals with very low level implementation details; the principal concern of this chapter is developing hardware that will interpret Rete networks at high speed. The chapter first describes a number of changes that could be made to the OPS2 implementation; the most important of the changes is to adopt a new representation for nodes and data elements. The chapter then begins its consideration of hardware. It shows that a few inexpensive additions to the hardware of a conventional processor would allow it to interpret the nodes of the network as fast as it interprets the instructions of its native instruction set. Next this chapter presents some calculations which were made to evaluate the suggested hardware. These calculations show that OPS2 would run more than two orders of magnitude faster on this hardware than on the Lisp-based interpreter for OPS2; the calculations also show that OPS2 would run more than an order of magnitude faster on this hardware than on an interpreter that was hand coded in assembly language. The chapter ends with a brief discussion of executing Rete networks on parallel computers.

6.1 Eliminating References to Primary Memory

The first step in developing a fast interpreter for the network is to eliminate all possible references to the computer's primary memory. This section describes changes to the interpreter's representations that eliminate most of the references required to perform tests, to index into data elements, and to search the node memories.

6.1.1 Eliminating References During the Tests

The performance of a test by a one-input node often involves examining more than one word in primary memory. The reason for this is that the Lisp-style list is an inappropriate representation for data elements.

A cell in a Lisp list consists of one 36-bit word, divided into two fields of 18 bits. One of the fields (the CDR) holds the address of the next cell in the list, or a special termination pointer if the cell is the last one. The other field (the CAR) holds the address of the contents of the cell. The list **(A (B C))**, for example, comprises four words. The CAR of the first word points to the atom **A**; the CDR points to the next list cell. The CAR of the next list cell holds a pointer to the list **(B C)**; the CDR holds the list terminator. The other two cells of **(A (B C))** represent the sublist **(B C)**. The CAR of the first points to the atom **B**, and the CAR of the second points to the atom **C**.

Primary memory is divided into blocks for list cells and atoms, for integers and floating point numbers, for machine language instructions, and for a few other data types supported

by Lisp. Whether a field holds a pointer to an atom, a list, or something else is determined by comparing the numerical value of the pointer to the upper and lower bounds of the various blocks of memory. If the block pointed into holds more than one data type, then the word pointed to must be examined to determine which it is.

Consider what is necessary to perform a test at an &LEN node. After INDEX locates the pointer to the subelement to be tested, a test must be made to determine that the value of the pointer lies in the correct range for lists. Then the word pointed to must be tested to determine conclusively that it is a list rather than one of the other data types in that block. The cells in the list must then be visited one by one and counted, and finally the result must be compared to the constant stored in the node.

The tests performed by the nodes would require only one memory reference if two fields were added to each list cell. The first field would indicate the type of the datum pointed to by the CAR of the cell. When the CAR points to a list, the second field would hold a count of the number of list cells contained in the list. Testing the length of a list with this representation would involve examining the type field to determine whether the CAR points to a list, and if it does, comparing the length field to the constant stored in the node.

6.1.2 Eliminating References During Indexing

The indexing performed by the network interpreter is very primitive. Consider this condition element (A (=X =)(=X =)) and the &VIN node that is generated for it.

```
(&VIN (. . .) (VARIABLE 1 2 1) (VARIABLE 1 3 1))
```

Note that both the index vectors in this node contain only constants. This is true of all index vectors generated by the OPS2 compiler. The meaning of these indices is not unlike the meaning of Fortran or Algol array indices, yet while the OPS2 interpreter always locates subelements by counting, taking one memory access per count, the other languages convert array references like I[1, 2, 1] into direct accesses of the desired element. The following subsections discuss making the network interpreter equally efficient.

6.1.3 Storing Cells in Contiguous Locations

The reason that INDEX cannot directly access subelements in Lisp-style lists is simply that logically adjacent cells are not necessarily physically adjacent. The third element of an ordinary array can be found without visiting the first two elements because there is a known relationship between the locations of the elements. If each array element occupies one word in the computer's memory, the address of the third element is two greater than the address

of the first. Since the individual cells of a Lisp-style list can be located anywhere in memory, the third element of a list cannot be reached without examining the CDR fields of the first and second elements. The obvious change to be made to lists, then, is to store cells in contiguous locations like array elements.

If the representation for list cells suggested in section 6.1.1 is used, each cell contains three pieces of information: the CAR of the cell; the indication of whether the CAR points to a list cell, an atom, an integer, or something else; and if the CAR points to a list, the length of the list. Since the elements of the list are stored in contiguous locations, the CDR field can be eliminated. Suppose that the following list was to be stored beginning at location 0 in memory.

(A (P Q) (P Q))

The first cell would hold the description of the top level of the list. The type field of this cell would contain the indication for type list. The length field would contain 3. The CAR field would point to first element of the list; since the list is stored beginning at location 0, this field would hold a 1. The cell at location 1 would hold the pointer that represents the atom A. Its type field would contain the indication of type atom. The cell at location 2 would hold the description of the first list (P Q). The fields for this cell would be similar to those in location 0. The cell at location 3 would hold the description of the second sublist. The elements of the two sublists would follow.

	Type	Length	CAR
0	list	3	1
1	atom		A
2	list	2	4
3	list	2	6
4	atom		P
5	atom		Q
6	atom		P
7	atom		Q

The &VIN node for (A (=X =) (=X =)) would then become

(&VIN (. . .) (VARIABLE 4) (VARIABLE 6)).

One might ask how the compiler knew that the list was going to be stored starting at location 0. Two answers are possible. The first is that location 0 could be dedicated to the elements being tested by the one-input nodes; since the changes made to working memory are processed one at a time, the element being changed could be copied to location 0 before the match begins. The other answer is that the element is not really stored in location 0 and that the numbers 4 and 6 point not to absolute locations 4 and 6, but to 4 and 6 locations beyond the start of the element, wherever that is. A base register in the interpreter could be loaded with the address of the first cell of the list before the match begins.

6.1.4 Condition Elements Containing "!"

A property of this contiguous storing of cells is that different lists can sometimes appear very similar. Consider the following list.

```
(A (P Q P) (Q))
```

Its representation:

	Type	Length	CAR
0	list	3	1
1	atom		A
2	list	3	4
3	list	1	7
4	atom		P
5	atom		Q
6	atom		P
7	atom		Q

The tabular representation of this list is identical to the one in the last section except for cells 2 and 3.

The &LEN nodes render this effect harmless in most cases. The &VIN node from the last section for example

```
(&VIN (. . .) (VARIABLE 4) (VARIABLE 6))
```

would not have a chance to test (and erroneously accept) the above element because it is preceded by &LEN nodes. In networks like those constructed by the OPS2 compiler, testing of wrong elements cannot occur if a condition element contains no ellipsis character ("!").

But consider what can happen when the condition element contains "!".

```
(A (=X ! =) (=X ! =))
```

The &LEN and &LEN+ nodes for this condition element would admit both

```
(A (P Q) (P Q))
```

and

```
(A (P Q P) (Q)).
```

Because &LEN+ nodes are less selective than &LEN nodes, they are not always sufficient to prevent the confusing of elements. If the &VIN node for this condition element is to avoid confusion, it cannot access subelements directly using the representation of the last section.

This node (and other nodes in similar circumstances) can, however, access subelements using fewer memory references than it would with the current Lisp representation. To locate the first subelement of the third subelement, for example, the &VIN node would read 2 cells. First it would examine location 3 to determine where the third subelement begins. Then it would access that location -- location 6 in the first data element, 7 in the second -- to get the pointer contained in the CAR field. In the Lisp representation, 4 cells would be read. To locate the pointer to the third subelement would require following the CDR pointers of the first two cells. The third subelement (in location 3 above) would have to be read to get the value in the CAR field, and then the location pointed to by the CAR field read to get the subelement to test. Thus the representation of the last section halves the effort of accessing the subelement in this case.

A different representation, described below, would reduce the effort still further, allowing direct accessing of subelements in all cases.

6.1.5 Storing Cells in Dedicated Locations

The next representation involves setting a maximum length for all the subelements and a maximum depth to which lists can be nested. Suppose no list was allowed to contain more than 10 elements and that lists could be embedded to no more than two levels (see section

6.3.5 for a discussion of the limits). Then no data element would occupy more than 111 locations in the tabular representation. If the full 111 locations were allocated to each element, and if the cells to be left empty were carefully chosen, it would be possible to access any subelement in one step. The subelements should be stored as follows. Location 0 holds the cell that describes the top level of the list. Locations 1 through 10 hold the cells that describe the subelements just below the top level. If the element has less than 10 subelements there, the unneeded cells are left empty. Locations 11 through 20 hold the cells describing the subelements of the first subelement. Locations 21 through 30 hold the cells describing the subelements of the second subelement. Locations 41 through 110 are used for the subelements of the third through tenth subelements.

Since the sublists always occupy the same locations when they are present, the information that was held in the CAR field of the type "list" cells is no longer needed. If the cell in location 1 has "list" in its type field, then the list will necessarily begin in location 11. If the cell in location 2 has "list" in its type field, that list will begin in location 21. The CAR fields cannot simply be omitted since the information they hold for type "atom" cells is still needed; but by rearranging the information in all the cells, the number of fields can be reduced from 3 to 2. Cells pointing to lists require a type field and a length field; cells pointing to atoms require a type field and a CAR field. The length and CAR fields can be combined into one field -- called perhaps the value field.

In this new tabular representation, the data element

```
(A (P Q) (P Q))
```

is (only the occupied cells are shown)

	Type	Value
0	list	3
1	atom	A
2	list	2
3	list	2
11	atom	P
12	atom	Q
21	atom	P
22	atom	Q

The data element

```
(A (P Q P) (Q))
```

is

	Type	Value
0	list	3
1	atom	A
2	list	3
3	list	1
11	atom	P
12	atom	Q
13	atom	P
21	atom	Q

6.1.6 Choosing a Tabular Representation

In choosing between the representations in sections 6.1.3 and 6.1.5 one is faced with the common tradeoff of space efficiency for time efficiency. The representation in section 6.1.5 allows faster accessing of subelements, but it can be wasteful of space. This section proposes a compromise: using one representation for the one-input nodes and another for the rest of the nodes.

When the match is performed once for each change made to working memory (as it is in the OPS2 interpreter), all the one-input nodes test the same data element. Since many one-input nodes will be activated each time if the production system is large, it would be reasonable to convert the data element from its normal representation (presumably the representation of section 6.1.3) into the expanded representation before executing the nodes. The time saved in executing the one-input nodes could far exceed the time required to convert the data element. The extra space required would not be great because at any time only one data element would be in the expanded representation.

The typical two-input node would not be slowed excessively by this compromise. Even though all the elements they test will be in the compact representation, the two-input nodes

will have to use the slower form of indexing only on occasion. In performing the variable tests for

```
(A (=X ! =) (=Y ! =))
```

for example, the nodes will have to use the slower indexing for =Y, but they can index directly to =X. If future production systems use "!" as rarely as present day production systems, most of the tests will involve direct indexing.

6.1.7 Tokens

If the compromise described above is adopted, some node programs will have to be made more flexible. The reason for this is that one- and two-input nodes will then differ in a fundamental way. With this table in use, the one-input nodes will not send tokens to their successors; they will send only control, relying on the successor nodes to read the data element from the table. Since the two-input nodes must build tokens as before, they will pass both control and tokens to their successors. Because &P, &MEM, and &TWO nodes can follow either one- or two-input nodes, they must have the flexibility to receive either control alone or both control and a token. In addition, either the two-input nodes must have the same flexibility, or the &MEM and &TWO nodes must be able to build tokens from the data in the table.

6.1.8 Eliminating References During Node Memory Examination

Since most of the data parts stored in the node memories contain only one data element (see sections 4.2.3 and 5.4.2) it is particularly important that examining the single element entries not require the fetching of many words. Best is to store all the information for these entries in one word. A 36 bit word might contain one 17 bit field to index an element in working memory, an 18 bit field to link this entry to the next entry in the node memory, and a 1 bit escape field. The escape field is needed for the data parts that hold pointers to more than 1 data element. The pointers to the remaining elements could be stored 2 to a word in contiguous locations.

6.2 Other Efficiency Measures

The following sections contain a diverse collection of methods for speeding up the nodes. These methods are not as important as the representation changes described above, but they are worthwhile.

6.2.1 Different Memory Technologies

If data elements are to be copied before they are tested by the one-input nodes, as suggested in section 6.1.6, the interpreter could be made significantly faster by copying the elements into a high speed memory. While it might be too expensive to store all the working memory elements in a high speed memory, it surely would not be too expensive to store the one element being processed by the one-input nodes.

6.2.2 Faster Two-input Nodes

The measurements made in the last chapter indicate that when large production systems are executed, most of the two-input node activations will cause no tests to be performed. Usually when a two-input node is activated by the arrival of a token on one input, the node will examine the memory of the other input, find it empty, and passivate itself. This section describes how the check can be made much faster than it is in the OPS2 interpreter.

If each two-input node carried internally an indication of whether its input memories were empty, the network interpreter could avoid many unnecessary accesses of the memories. It would be necessary to add two flag bits to each two-input node (one bit for each input memory). A FALSE bit would indicate that the corresponding memory was empty; a TRUE bit, that the corresponding memory was non-empty. Before execution of the production system began, all the bits would be set to FALSE since all the node memories would be empty. Then during execution, the network interpreter would update the flag bits of each node when the node was activated. When a token with a VALID tag arrived on one of the inputs, the bit for that input would be set to TRUE. The arrival of an INVALID token would not in itself be sufficient reason to set the bit FALSE; the memory might have contained more than just the one data part. But since the interpreter would have just examined the memory, it could easily remember whether the memory was emptied and set the bit accordingly.

Not fetching the node memories makes the interpreter faster because it reduces the number of memory references. If the two-input nodes are too long to be fetched in a single memory access, the reduction in the number of memory accesses can be more significant. The flag bits could be stored in the first word of the node. The interpreter could examine the bits as soon as they arrived, and if the appropriate bit was FALSE, it could abort the fetching of the remainder of the node.

6.2.3 Faster Tests at Two-input Nodes

The two-input node programs listed in Chapter 3 perform some operations more times than they should. In those programs, every time through the inner loop, INDEX is called to extract both the data subelements to be compared. Since one of the data parts indexed into remains unchanged throughout the activation of the node (this is the data part of the token that just arrived at the node) its subelements could be extracted before the loops were entered and stored in local variables. If the node had many tests to perform, this would reduce the number of calls to INDEX by almost half.

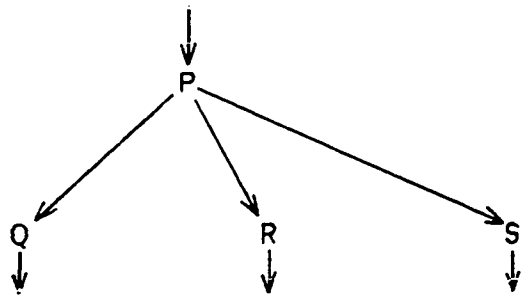
6.3 Bit Vector Nodes

If computers are to be able to interpret nodes at high speed, the nodes must have an appropriate representation -- bit vectors. In addition, the nodes must not be longer than necessary, or memory cycles will be wasted simply fetching the nodes. The following sections describe a representation that allows the one-input nodes to be held in one 36-bit word. The representation is somewhat wasteful of bits, but it is simple and therefore more suitable for exposition than a maximally efficient representation would be.

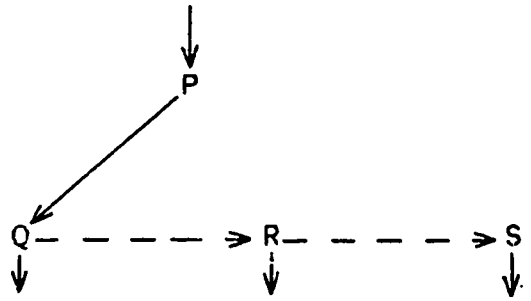
6.3.1 Fixed Length Successor Fields

This section shows how the successor fields in the nodes could be made to have a fixed length.

The technique used is a standard one for storing trees. Consider a node P which has three successors, Q, R, and S.



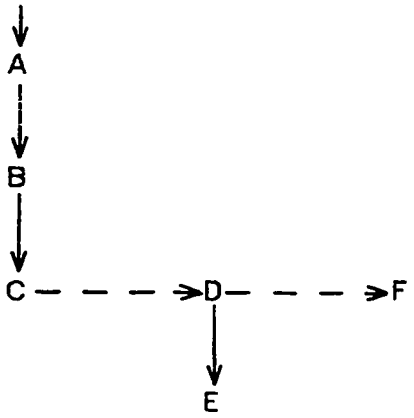
Instead of storing three pointers in node P, the compiler could store one pointer in P to Q (say) and a new kind of pointer -- a "brother" pointer -- from Q to R and from R to S. S would hold an indication that it is the last of the brothers; a null brother pointer would probably be used.



In this way, the variable length successor field in each node is replaced by two fixed length (length one) fields, one field for a pointer to a successor, and one for a pointer to a brother node.

6.3.2 Eliminating the Successor Field

Of the two pointers, successor and brother, one can be made implicit and eliminated from the node. Suppose the successor field is to be eliminated. If the compiler arranges the network so that one successor of each node is adjacent to the node, the interpreter can locate that successor by adding the appropriate offset to the ordinal position of the node. (The offset is the length of the node.) In the following network, for example, nodes A, B, and C would be adjacent, as would D and E. Nodes C, D, and F would not necessarily be adjacent.



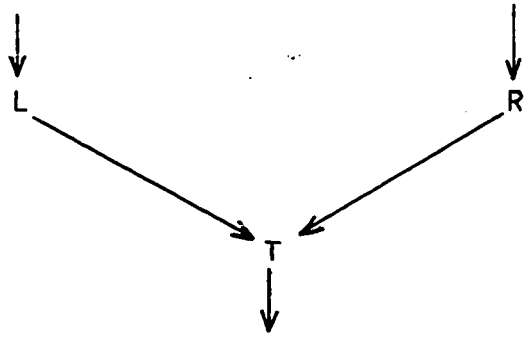
The two-input nodes present a problem; since they have two immediate predecessors, they cannot be adjacent to both. One solution to this problem is to store explicit successor links in all memory nodes (&VEX and &NOT nodes are always preceded by &MEM or &TWO nodes).

6.3.3 Short Successor and Brother Fields

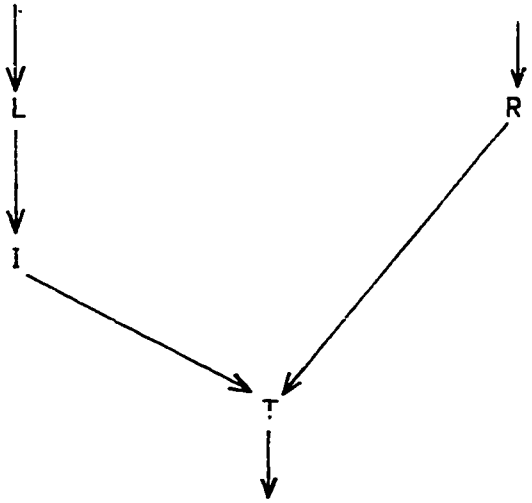
It would be unwise to make the successor and brother fields large enough to allow one node to point to any other node in the network. If very large production systems are compiled, networks containing 100,000 or 1,000,000 nodes or more may be built. If the brother fields were to be able to point to any of the other nodes, the fields would have to contain more than 20 bits. A brother field 20 bits long might be acceptable if 48 bits could

be allocated for a one-input node, but if the one-input nodes are to be made 32 or 36 bits (to fit into one word of a contemporary computer) fields of more than about 12 bits cannot be used.

These short fields can be used if two techniques from minicomputer design are adapted. First, the pointer must be interpreted not as an absolute address, but as an offset to be added to the address of the node containing the pointer. Second, an escape mechanism must be included to allow longer pointers when they are necessary. One way to achieve this is to define a new node type which performs no processing and which therefore has unused bits that can implement a long pointer field. For an example of how this node would be used, suppose that L and R are two memory nodes which have the same successor, T; and suppose that L and R are located so far apart that, regardless of where T is placed by the compiler, L and R cannot both point to it using their short successor fields. With the indirect pointer node available, the compiler could place T close to one of the nodes -- say R -- allowing R's successor field to point directly to T, and it could use an indirect pointer between L and T. The indirect pointer, I, would of course be placed close to L. The fragment of the network, which in OPS2 would be



would become



6.3.4 Short Fields for Constants

In OPS2 it requires 18 bits to store a constant in an &ATOM node. If the Lisp representation was abandoned, and integers used to represent atoms, fewer bits would suffice. To represent K atoms would require only $\log_2(K)$ bits. The number of bits to allow is

difficult to predict without experience using very large production systems, but perhaps 10 bits (for 1024 atoms) is reasonable. Consider that the artificial languages Basic English, which contains about 850 words, and Basic French, which contains about 1000 words, are sufficient for much of day-to-day human discourse [40], and that computer programs operate in a world which is impoverished indeed compared to that of humans. Certainly the practice of renaming goals and data could result in many made-up words (Move1, Move2, and so on) but these need not be represented internally as individual words. The interpreter could automatically convert elements like

```
(Move3 Block1 to Position4)
```

to a form that does not use made-up words; for example,

```
((Move 3) (Block 1) to (Position 4))
```

or

```
(Move 3 Block 1 to Position 4).
```

If the production system is to allow integers as well as literal atoms, the constant field will have to be longer than 10 bits. A field of 12 bits might be reasonable; it would allow using 1024 atoms plus integers in the range [-1536, 1535].

6.3.5 The Length of the Table for One-input Nodes

Three factors should be considered in choosing the size of the table for the expanded data elements. First, of course, is the length of the data elements. The table need not be big enough to store the longest elements; if long elements are rare, they can be split into several shorter elements. (If the interpreter performs the splitting, the user need not become aware that the system does not directly support long elements.) The second consideration is the expense of the indexing operations performed by the two-input nodes. For the sake of simplicity, it would be preferable to use the same index for a given subelement in both the one-input and two-input nodes. If the table size is chosen inappropriately, interpreting the integers with the compact (non-expanded) data representation could involve performing integer divisions. The third consideration is the length of the the nodes' index fields; the the fields must be short.

A possible compromise for the three factors is to allow a data element to have 15 subelements, each of which can be an atom or a list of up to 16 atoms. After having seen the

measurements in the last chapter, the reader might question why so much space has been allocated to store one data element. The answer is that this format is designed to support attribute-value data elements as well as list data elements.[1] Assuming one of the 15 subelements is used for the name of the object being described, this element format would allow 14 atiributes, each with a value about as complex as a typical OPS2 data element. If 14 attributes were not enough to describe some object, multiple data elements could be used. Two data elements would allow 28 attributes, three would allow 42, and so on. This format meets the requirement that the index fields must be short. The table for the expanded representation would have 256 slots, and thus only 8 bits would be required for the index fields. Finally, since the maximum length allowed for the subelements is a power of 2, the indices can be interpreted cheaply with the non-expanded representation of data. The algorithm is

```
1. If the INDEX is 0, return the pointer to the top
level of the list.
2. If the INDEX is greater than 0 and less than 16,
return the first level element of that number.
3. Otherwise,
        Begin
        4. Set HIGH = the high order 4 bits of INDEX.
        5. Set LOW = the low order 4 bits of INDEX.
        6. Use HIGH to select one of the first level subelements
        of the element.
        7. Use LOW to select one of the subelements of the
        embedded list, and return that.
        End
```

6.3.6 The Type Field of the Nodes

The networks constructed by the OPS2 compiler contain ten node types: &BUS, &ATOM, &LEN, &LEN+, &USER, &VEX, &NOT, &MEM, &TWO, and &P. The &BUS node could be eliminated, but the other nine are necessary. To implement the binary search for constant names discussed in Chapter 4 would require two more node types, one to test for constants names less than some fixed quantity and one to test for constants greater than a fixed quantity. And finally, one node type is needed for the indirect pointer nodes. This gives a

[1]Appendix II shows how a collection of name-attribute-value triples can be represented as a single data element with one subelement dedicated to each attribute. Note that this representation could be used internally even if a different external representation were used.

total of twelve node types.[2] The nodes therefore need four bits in their type fields.

6.3.7 Bit Vector One-input Nodes

The one-input nodes contain four fields: a type field, a field for the pointer to the node's brother, a field for the pointer into the INDEX table, and an argument field. The argument field of an &VIN node will contain another pointer into the INDEX table. The argument field of an &USER node will contain a pointer to the predicate to be evaluated. The argument fields of the other one-input nodes will contain constants to be tested for. These fields can be made to fit into one 36-bit word. The length of the type field has already been set at 4 bits, the length of the argument field at 12 bits, and the length of the index field at 8 bits. This leaves 12 bits for the brother field, enough to point to any one of 4096 nearby nodes. The one-input node format is then

Type	4 bits
Brother	12 bits
Index	8 bits
Argument	12 bits

6.3.8 Bit Vector Indirect Pointer Nodes

The indirect pointer nodes contain only two fields, a type field and a long pointer field. If the type field is the usual 4 bits long, the pointer field could contain up to 32 bits, though the interpreter might ignore some of them.

Type	4 bits
Pointer	32 bits

6.3.9 Bit Vector Two-input Nodes

A two-input node has seven fields, excluding those containing information about the variables it tests. Two of these fields already have their lengths fixed. The type field is 4 bits long, and the field indicating whether the memories are empty (the memory status field) is 2 bits long. Four of the remaining five fields in the node are pointers to other nodes: two

---

[2] If the hash table nodes were used instead of the binary search nodes, eleven node types would be needed. It will be assumed in this chapter that the binary search nodes are used because use of the hash table nodes would complicate the calculations presented later in the chapter.

pointers to the memory nodes it reads[1], a pointer to its left brother, and a pointer to its right brother. Since these four pointers certainly cannot be held in 30 bits, the fixed part of the two-input nodes must be at least 72 bits long. In order to make best use of the memory status bits, the pointers should be no longer than 15 bits. When the interpreter executes a node with a zero in the appropriate memory status bit, it will use only three fields: the type field, the memory status field, and one of the brother fields. If the pointers were 15 bits long, all the fields could be held in the first word of the node, and the interpreter could avoid fetching the remaining words. Four 15-bit pointers plus the type field and the memory status field require 66 bits. This leaves 6 bits to indicate the number of variables tested, a number that should be ample. The field that holds this count is called the continuation. The fields of the two-input node, excluding those for the variables, are then:

Type	4 bits
Memory Status	2 bits
Left Brother	15 bits
Right Brother	15 bits
Left Memory	15 bits
Right Memory	15 bits
Continuation	6 bits

For each variable it tests, a two-input node must have two indices plus an indication of which test to apply. Each index field must hold 8 bits to access subelements, 1 bit to indicate whether to use these 8 bits in the fast or slow addressing mode (see section 6.1.6), plus enough bits to indicate which element of the data part to access. Since LHSs containing more than 16 condition elements are rare, 4 bits for this should be enough, but more bits would be desirable. If 5 bits were allocated, the fields would be:

Variable Type	8 bits
Left Index	14 bits
Right Index	14 bits

[1]The &NOT node will not store its own tokens; it will rely on a memory node.

6.3.10 Bit Vector Memory Nodes

Memory nodes contain five fields: two successor pointers (see section 6.3.2), a type field, a brother field[1], and a pointer into the space where tokens are stored. (Tokens and nodes surely will not be stored in the same address space in a reasonable implementation.) The type field is 4 bits, as always. The three pointers to other nodes can be 15 bits, as in the two-input nodes. If two words are allocated for a memory node, 23 bits are left for the pointer into token space. This should be far more bits than are needed; even assuming the 4 tokens per production growth rate seen in KERNL1 with the "retry" productions (see section 5.4.2), 23 bits would be sufficient for 2,000,000 productions.

Type	4 bits
Brother	15 bits
Left Successor	15 bits
Right Successor	15 bits
Tokens	23 bits

6.3.11 Bit Vector Production Nodes

The production nodes need contain only two fields: a type field and the name (or number) of the production. If the type field is the usual 4 bits, the name field could contain up to 32 bits:

Type	4 bits
Name	32 bits

6.4 A Machine to Directly Interpret Rete Networks

This section is the heart of Chapter 6. It demonstrates the usefulness of the representations described in the first three sections. It argues that these representations make the task of interpreting nodes so simple that a very slightly modified conventional processor could interpret nodes as fast as it interprets conventional instructions.

---

[1]The brother field is not strictly necessary. Since memory nodes do not test or modify the tokens they receive, they could "adopt" their brothers, making them their immediate successors. Keeping the brother pointers was deemed simpler, however.

6.4.1 Comparing One-input Nodes to Conventional Instructions

Before discussing the special hardware that is needed to interpret Rete networks, it is worthwhile to compare nodes to the instructions of conventional computers. The comparison between a node and a conditional branch instruction is most instructive.

The PDP-10 instruction set includes an instruction with the mnemonic name "CAIN" (Compare Accumulator Immediate and skip if Not equal) which is much like an &ATOM node. A CAIN instruction contains an 18 bit constant and a register designation. It compares the constant to the contents of the register, and if the two are not equal, it causes the machine to skip the following instruction.

In its gross characteristics, the &ATOM node is similar. Both have constant fields. Both have fields that point into blocks of high speed registers. Both allow the following instruction (node) to execute only if the constant field is equal to the datum held in the register.

The differences between the two are principally detail differences. The CAIN instruction contains more fields, though the fields other than the ones holding the type, register, and constant are seldom used. The fields in the CAIN instruction do not have the same lengths as the fields in the &ATOM node. The &ATOM node has one implicit and one explicit pointer to other nodes, while the CAIN instruction has two implicit pointers to other instructions. The one difference between the two which cannot reasonably be called a detail is that the &ATOM node sometimes allows both the nodes it points to to be executed.

Similar comparisons (with similar results) could be made for the other one-input nodes. The &VIN node is like a compare of two registers -- achieved with the "CAMN" (Compare Accumulator to Memory and skip if Not equal) instruction on a PDP-10. With the representation of data elements suggested in section 6.1.1, the &LEN node is like the CAIN instruction; the &LEN+ node is like the instruction called "CAIL" (Compare Accumulator Immediate and skip if Less).

6.4.2 Special Hardware for the One-input Nodes

Since one-input nodes are similar to some existing instructions, most of the hardware necessary to interpret the nodes should already be present in a general purpose computer. In order to determine what other hardware might be necessary, this section examines every difference between the one-input nodes and the compare instructions.

The first difference is that the one-input nodes require a larger block of high speed registers. If the tabular representation suggested in section 6.3.5 is adopted, 256 registers

are required, compared to the 8 to 16 that are commonly used in a general purpose computer.

That a one-input node contains fewer fields and contains fields of different lengths would not make any special hardware necessary, provided the conventional instructions were interpreted by a general microprocessor. Microprocessors that are designed to interpret more than one instruction set usually contain flexible hardware for field extraction.

That a one-input node contains an explicit pointer to its brother might make some special hardware necessary. After this field is extracted from the word, it must be tested to determine whether it is null, and if it is not, the address of the brother must be computed from the field contents plus the address of the node being executed. One could program this on a microprocessor, but it would be faster if special hardware were available.

The final difference, that it is sometimes necessary to activate both successors of a node, makes it necessary to maintain a stack of node addresses. Clearly when both successors are to be activated, one address must be stored temporarily. The reason for using a stack is to minimize the amount of information that must be stored at any time. If nodes are executed in a depth-first manner (i.e., activating the successor of a node before the brother) the number of addresses stored can never exceed the number of nodes in the longest path through the network. Since the length of the longest path depends on the complexity of the individual LHSs, and not on the number of LHSs, this stack need not be large. It could therefore economically be held in a high speed memory. One might want to include special hardware to maintain the stack -- something to push addresses onto the stack and something to watch for stack overflow.

6.4.3 Special Hardware for the Two-input Nodes

No special hardware would be required to interpret the two-input nodes. The necessary hardware -- for field extraction, for bit testing, and for computing and stacking brother pointers -- must already be present to interpret the one-input nodes.

Neither is special hardware necessary to perform the tests at the two-input nodes. The functions of extracting fields, accessing primary memory, and comparing integers would be provided by any general microprocessor.

One piece of hardware would be necessary, however, to allow control to pass between the two-input nodes and their successors. If the OPS2 algorithm is adapted for the machine, the two-input nodes will require another small block of high speed registers for their use. Each incarnation of a two-input node maintains a few local values (see section 6.2.3). When a node

is suspended after outputting a token so that its successors can process the token, these values must be stacked. This stack can be small since most LHSs contain only a few condition elements and and a few variables. (The stack need not be large enough to handle every possible case; parts of the stack can be moved to primary memory on the extraordinary cases.)

6.4.4 Special Hardware for the Memory Nodes

No special hardware is required for either the memory nodes or the tests performed by the memory nodes. Interpreting the nodes themselves involves no operations not also needed to interpret the one- and two-input nodes. The same is true of the fetching and testing of elements in the memories. The task performed by a memory node, maintaining a linked list in primary memory, requires only the ability to read from and store into primary memory.

6.4.5 Special Hardware for the &P Nodes

The &P nodes are executed so infrequently that special hardware seems unjustifiable.

6.4.6 Special Hardware: Conclusion

In summary, the only special hardware needed is (1) a quantity of high speed memory; (2) the hardware to interpret the brother fields; and (3) the hardware to maintain a stack. To perform the last two functions it is necessary to have a comparator to determine if the brother field is null and a register that can be incremented and decremented by one (to hold the pointer to the top of stack). No adder would be needed to compute the address of the brother; it would be sufficient simply to concatenate the high order bits of the current address to the brother field. The high speed memory is needed for the INDEX table for the one-input nodes, for the stack of pending nodes, and for the stack of the two-input nodes' locals. Section 6.3.5 discussed the length of the table for INDEX and concluded that 256 words should be sufficient. If the LHSs of productions are as complex as those in the three measured production systems, 64 words for each of the stacks should be ample.

The high speed registers are the only items in the list that are important. If the high speed registers were not available, the number of references to primary memory would increase several fold. If the other hardware were not available, a few more microinstructions would be executed in interpreting each node.

There was, however, a reason for including the hardware: to allow arguing that, with the special hardware, the processor would be able to interpret one-input nodes as fast as it

interprets conventional compare instructions. With the hardware, the steps involved in interpreting the two are nearly identical: extracting a constant from a field in the instruction; retrieving a value from a high speed register; comparing the two; and then choosing which of two instructions to fetch next based on the outcome of the comparison. In one case, the choice is made by conditionally adding one to the program counter; in the other case the choice is made by conditionally loading the program counter with the value in the top of the node address stack. But surely neither of these is appreciably slower than the other. The computing and stacking of the brother address, which has no analogue in the interpretation of the compare instruction, is performed in parallel with the other steps. Thus unless the hardware for the stack is quite slow, it will add nothing to the time required to interpret the node.

6.5 Estimating the Performance of a Rete Machine

This section contains some calculations that can be used to evaluate the hardware just described. Subsection 6.5.1 contains an estimate of the relative speeds of four machines which have varying degrees of hardware support for the match algorithm. The rest of this section is devoted to calculating how fast a few representative production systems would run on machines with all the hardware described above.

6.5.1 The Time Required to Execute a One-input Node

Calculations are made in this subsection to determine how fast a KL10-sized processor could execute four implementations of one-input nodes. Considered are (1) the OPS2 Lisp implementation, (2) an assembly language version using the tabular representation of data elements, (3) a microcoded version using the same representation, and (4) a processor with the special hardware described above. The reason for considering only one-input node times is that these times have predictive value. Since the one-input nodes will account for ever more of the total execution time as production systems grow larger, the relative times computed here are reasonable estimates of the ultimate relative times for the entire match.

The measurements reported in Chapter 5 were made on a KL version of a PDP-10. Since this machine has a cache, to compare single node times on this machine would be difficult and potentially misleading. The time required to execute a node could vary by several hundred percent depending on whether the node happened to be in the cache. This problem will be avoided by hypothesizing a machine which runs Lisp as fast as a KL10, but which does not have a cache. This hypothetical machine should have a memory bandwidth of 2.1 million

words per second, and it should be able to execute 1.25 million instructions per second.[1]

The speed of the Lisp version can be estimated from the data in Chapter 5. Including the time for one call to INDEX, the three production systems measured in Chapter 5 required 346 microseconds (KERNL1), 316 microseconds (HAUNT), and 326 microseconds (PH-632) to execute a one-input node. The average of these three is 329 microseconds.

To estimate the speed of the assembly language version, code was written for the part of the interpreter needed to execute &ATOM nodes (see Appendix III). The time required to execute the instructions on a KA10 processor was calculated from published information about the processor [15], and the time was adjusted to reflect the higher speed of the hypothetical machine. The result is that the node would take about 11 microseconds to execute.

If the microprocessor of the hypothetical machine has the usual field extraction and comparison capabilities, the microprogrammed interpreter should be able to execute the nodes at memory speed. Executing a one-input node requires 3 memory references (1 to pop the node's address off of the stack, 1 to fetch the node, and 1 to fetch the data subelement tested by the node). Thus an &ATOM node would require about 1.4 microseconds to execute.

As argued in section 6.4.6, a one-input node should be as fast as a conventional compare instruction if the special hardware is available. Executing a CAIN instruction on a KA10 takes about 2 microseconds. Therefore, on the hypothetical machine a one-input node would take about 0.5 microseconds. Note that the same value results from counting memory references; executing a one-input node requires one memory reference, or about 0.48 microseconds on the hypothetical machine.

The following table summarizes the comparison made in this section.

---

[1]These values were determined as follows. Experiments were run which showed that Lisp on a KL10 is 4.2 times faster than Lisp on a KA10. Published results of KA10 performance measurements indicate that it has an effective processor-memory bandwidth of about 0.5 million words per second and that it can perform about 0.3 million instructions per second [23]. Multiplying these values by 4.2 gives 2.1 million words per second bandwidth and 1.25 million instructions per second execution speed.

	Time	Relative Time
Microcode Plus Special Hardware	0.5 μsec	1
Microcode	1.4 μsec	3
Assembly Language	11 μsec	20
Lisp	329 μsec	700

Table 6.1. One-input Node Implementations

6.5.2 Time Costs of the Nodes

The remainder of this section is devoted to calculations of the total match times for several typical production systems running on the hardware described above. Before these calculations can be made, it will be necessary to estimate the time costs of the various operations performed by the network interpreter. Making these estimates will not be difficult if one assumption is allowed: that the speed of the machine depends directly on the number of primary memory accesses made. This seems a reasonable assumption; since the processing performed by the nodes is simple, a processor of even moderate power would probably be limited by memory bandwidth.

If the special hardware described above is available:

- Activating a one-input node requires one memory reference (to fetch the node).

- Activating a two-input node requires one memory reference when the memory status bit is false. This should be the most common case in large production systems. But when the bit is true, one more reference must be made to fetch the second word of the node, one reference for each variable-describing part of the node, one reference to extract the memory pointer from the memory node, and at least one reference for each variable to extract the subelements from the token.

- Activating a memory node requires two memory accesses, excluding the ones necessary to search and modify the node memory. Searching the node memory is accounted for in the cost of the memory tests. Modifying the memory, which is necessary only if the token is tagged VALID or INVALID, might require two more memory accesses if the free words are kept on a linked list.

- Performing a test at a two-input node requires one memory reference to fetch

the data subelement unless the slow form of indexing is used. Since most condition elements do not contain !, fast indexing should be more common. If the typical two-input node tests one variable, following the links in the memory node's list of data parts will add one memory access per test. If the two-input node tests more variables, following the links will add proportionately less overhead.

- Performing a test at a memory node requires one memory reference if the data parts contain only one data element.

The &P nodes are not mentioned because the processing performed by those nodes is not a significant part of the total.

6.5.3 KERNL1 on the Rete Machine

The following is an attempt to estimate how fast the Rete machine would perform the match for the 346 production version of KERNL1 (the slowest production system measured in Chapter 5).

Table 5.6 shows that processing the average action involved 78.8 activations of one-input nodes, 23.8 activations of two-input nodes, 71.5 tests performed by the two-input nodes, 12.5 activations of memory nodes, and 170 tests performed by the memory nodes.

The number of memory references during the match can be estimated from the information given in the last section. To make the estimate conservative (i.e., almost certainly an overestimate)

- Assume that every test performed by a two-input node uses slow indexing. Since the average data element contains just over 1 nested list (see table 5.3) estimate that indexing requires 2 memory accesses. Performing the test thus requires 3 accesses (1 to follow the links in the node memory plus 2 for the indexing).

- Assume that 3.8 of the two-input node activations involve performing tests. The number cannot be greater than 3.8 because the same 71.5 tests were performed when only 3.8 nodes were activated (the run of 56 productions). Since the average two-input node performs about 1 variable test, activating one of these two-input nodes requires 6 accesses (2 for the node itself, 1 for the variable part of the node, 1 to reach the memory node, and 2 to index into the data part of the token).

- Assume that every node is tagged VALID or INVALID, and that 2 memory references are required to take a word from the free storage list or to put a word onto the free storage list. Then activating a memory node requires 4 accesses.

The total number of memory accesses is then

556 = 78.8 * 1	(One-input nodes)
+ 20 * 1 + 3.8 * 6	(Two-input nodes)
+ 71.5 * 3	(Two-input tests)
+ 12.5 * 4	(Memory nodes)
+ 170 * 1	(Memory tests)

On a Rete machine with an effective processor-memory bandwidth of 2.1 million words per second (the equivalent of a KL10) the match for KERNL1 would require about 265 microseconds. The match required 111 milliseconds in OPS2, so the Rete machine would be more than 400 times faster.

This increase in speed would be accompanied by a substantial reduction in the size of the compiled LHSs. The 346 production system had about 1675 one-input nodes, 346 two-input nodes, 410 memory nodes, and 346 &P nodes. Assuming that the average two-input nodes tests 0.9 variables (this is the figure from section 5.3.4) then with the node format described in this chapter, the nodes for these 346 productions would occupy about 3800 words. In the OPS2 format, these nodes occupy about 16,600 words. The new format thus requires less than one-fourth as much space. This new format also requires iess space than the uncompiled form of the LHSs. Uncompiled, these LHSs consume about 8700 words.

6.5.4 A Larger Production System

Since it will eventually be necessary to run much larger production systems than KERNL1 or HAUNT or PH-632, this subsection estimates the time and space costs of executing these larger production systems. It is assumed here that the productions in the system are about as complex as those in KERNL1, because this allows the results of the last subsection to be used.

The space costs are easy to compute since the size of the network is a linear function of the size of the production system. The 346 production version of KERNL1 required about 3800 words for its network, so each production required about 11 words. Thus if a production system contained 10,000 productions, its network would consume about 110,000 words. If a production system contained 100,000 productions, its network would consume about 1.1 million words.

Determining the time costs of the larger production systems is less easy, since, as Chapters 4 and 5 showed, the costs of the different match operations scale differently. One simple, but slightly pessimistic way to resolve the problem of scaling is to assume that the number of two-input tests does not increase and that all other operations increase with the logarithm of the number of productions. The count of memory references made in the last subsection can

then be used to estimate time costs. Of the 556 memory references needed to perform the match for the 346 production system,

$$3.8*6 + 71.5*3 = 237$$

were performed to effect the two-input nodes' tests, and the remaining 319 were performed for other reasons. The cost of the match (measured in memory accesses per RHS action) for a system with P productions could therefore be estimated by

$$237 + 319\log_2(P / 346).$$

If the system contained 10,000 productions, this expression would evaluate to about 1800.

It is worth noting that in Chapter 1 it was estimated that a production system would have to perform about 1000 RHS actions per second to equal Lisp on a KL10. Since the KL10 has a processor-memory bandwidth of 2.1 million words per second, it appears that a KL10-equivalent machine with the extra hardware described in this chapter could execute a rather large production system at this rate or faster. A system with 10,000 productions, for example, would require about

$$1800 * 1000 = 1.8 \text{ million}$$

memory references for the match, leaving about 0.3 million references per second for conflict resolution and the RHS actions. If the RHS actions were compiled, this would be quite adequate for a production system like KERNL1.

6.5.5 Production Systems on Minicomputers

Since there is some interest in running production systems on minicomputers (see, for example [3, 4]), this subsection contains a estimate of the execution speed of a minicomputer with the production system hardware.

The calculations will assume that the computer is a PDP-11/40 [16], a microprogrammable machine, but one which has rather modest performance by today's standards. The PDP-11/40 is a 16-bit machine, and it has a processor-memory bandwidth of about 0.9 million 16-bit words per second. These 16-bit words are less than half the size of a PDP-10 word, but since the node formats of this chapter are not particularly efficient in their use of bits, two PDP-11 words could probably be used in place of one PDP-10 word.

On a PDP-11/40, then, the 346 production version of KERNL1 would require

$$2 * 3800 = 7600$$

words for the network, and processing each RHS action would entail about

$$2 * 556 = 1112$$

memory references. Since the PDP-11/40 has a bandwidth of about 0.9 million words per second, even if 20% of its time was spent evaluating RHSs and performing conflict resolution, it should still be able to perform more than 600 actions per second.

The principal limitation of the PDP-11 is that it has a small address space. It is difficult to have a data structure that extends over more than 32,768 words. Since each production's LHS would compile into about 22 words, this would be enough for only about 1500 productions. Using the same scaling assumptions as the last subsection, a system of 1500 productions would require about 1800 memory accesses per RHS action. An 11/40 sized machine should thus be able to execute a 1500 production system at at least 400 actions per second.

Finally, since a machine with microcode but no special production system hardware would run about 3 times slower, an unmodified PDP-11/40 should be able to run the 1500 production system at more than 130 actions per second and the 346 production system at more than 200 actions per second.

6.6 Parallelism

With the hardware enhancements suggested in this chapter, an otherwise conventional processor could perform tests at one- and two-input nodes as fast as it could execute conventional instructions. This section is concerned with the possibility of obtaining still faster execution.

First it should be pointed out that a few of the techniques commonly used by computer architects to speed up uniprocessors can be used for the network interpreter. Since the Rete Match Algorithm is memory limited, none of the techniques for speeding up the processor alone is useful. Thus the algorithm is not amenable to techniques like putting more data paths in the machine, pipelining the processor, or using special functional units. The techniques for achieving high processor-memory bandwidths are certainly potentially useful, however. For example, the measurements made in Chapter 5 indicate that making the memory wider would improve the performance when very large production systems are run. Adding a cache might also improve performance, but, as mentioned in Chapter 5, this cannot be determined conclusively until larger production systems can be measured.

While some of the uniprocessor techniques are feasible, the performance of the Rete Match

Algorithm could more easily be improved through the use of parallelism. For instance, there exist two reasonable ways to use low degrees of parallelism. The first is to store the network in a memory that can be accessed by all the processors, and to allow the processors to fetch and execute nodes in parallel. The second is to execute many small networks in parallel. In this method, a production system is divided into a number of smaller systems, and each of the smaller systems is compiled into a separate network. One processor is assigned to each network. The advantage of this method is that the overhead of synchronizing the processors would be small; synchronization would be needed only when the processors tried to update the conflict set. A disadvantage is that sharing of nodes is reduced, resulting in more space being used to store the network, and more operations being performed on each cycle.

To use the very high levels of parallelism promised by VLSI (very large scale integration) another technique is necessary. With continuing advances in semiconductor technology, computers containing thousands, or even millions, of processors may be feasible in a few years. Such computers would be tremendously powerful; assuming the processors are only a few times faster than today's one-chip computers, one million processors could perform a trillion operations per second. Programming these computers, however, will be difficult. Besides the obvious difficulty of dividing a problem into enough small units, the programmer will probably be faced with severe limits on communication. Almost certainly it will not be possible to send information between arbitrary pairs of processors cheaply (consider Illiac IV, and its problems with only 64 processors). There is a way to use the Rete Match Algorithm that avoids both of these problems. The production system could be compiled into a single network as in the serial processor implementations. The network could then be divided into regions containing only a few nodes each, and the regions assigned to different processors. With this organization each processor would communicate with only a few others (those whose nodes were successors or predecessors of its nodes). The organization obviously has the potential for extrememly high degrees of parallelism; the limit would be one processor for each node in the network.

7. Conclusions

The goal of this research has been to develop means for interpreting large production systems efficiently. The research has considered both the algorithms used by the interpreter and the hardware on which the algorithms run. The following section reviews the body of the thesis, discussing the contribution of each chapter to the goal of efficiency. The last section then presents a few suggestions for further research.

7.1 Summary of Previous Chapters

Chapter 1 introduced the problem attacked by this research: how to make pattern directed invocation of productions more efficient. Current production system interpreters often spend more than nine-tenths of their time evaluating patterns for this purpose, and since production systems are growing larger, the fraction of the time spent in evaluating the patterns is increasing.

Chapter 2 described the Rete Match Algorithm, an algorithm for performing this pattern evaluation. The chapter began by defining two properties of the match that make it possible to design efficient algorithms. The first of these, structural similarity, is the property that much commonality exists among the various patterns. The second, temporal redundancy, is the property that the working memory of a production system generally changes slowly from one cycle to the next. The Rete Match Algorithm takes advantage of structural similarity by compiling all the patterns into a single network of tests, with sharing of subnetworks used when possible. It takes advantage of temporal redundancy by storing all the information about matches and partial matches of productions from one cycle to the next. The algorithm described in this chapter can be implemented on both serial and parallel computers, and it is suitable for a broad class of production systems.

Chapter 3 described the OPS2 interpreter, the most recent interpreter to use the Rete Match Algorithm. The interpreter's programs and the data processed by the programs were described in enough detail to allow the reader to reproduce the interpreter. The algorithm in Chapter 3 is more efficient than the one in Chapter 2 because it has been specialized for one language (OPS2) and for implementation on a serial machine. The two changes that were necessary to effect this specialization are independent, so the reader should be able to infer how to construct another interesting interpreter: one for a wide class of languages, but specialized for a serial machine.

Chapter 4 contained an analysis of the OPS2 algorithm. This analysis determined how the time and space costs of the algorithm depend on the number of productions in production memory and the number of data elements in working memory. Expected dependencies were

determined as well as the usual best and worst case dependencies. To determine the expected dependencies it was necessary to examine the programming conventions used in the construction of large production systems. It was shown that two of the conventions strongly affect the efficiency of the interpreter: (1) putting a condition element that matches goals first in the condition part of each production, and (2) using RHS actions that encode information so that other productions cannot recognize it. The results of the analysis are shown in tables 4.1 and 4.2 on page 106. In the course of performing the analysis it was discovered that the OPS2 algorithm was not properly designed to take advantage of the above programming conventions, and that this improper design caused the expected time cost to be a linear function of the number of productions in the system (i.e., $O(P)$). Two simple modifications of the algorithm were described, either of which would reduce the dependency from $O(P)$ to $O(\log_2(P))$. Chapter 4 showed that $O(\log_2(P))$ is the best that can be achieved by any algorithm.

Chapter 5 described the results of experiments that were performed on the OPS2 interpreter while it was running the three largest OPS2 production systems. These production systems contained, respectively, 316 productions, 381 productions, and 1017 productions. One group of experiments were performed to determine the effects of production memory and working memory sizes. The results verified the analysis of Chapter 4, and produced numbers to supplement the order results of Chapter 4. Another group of experiments were performed to determine whether the algorithm would execute efficiently on computers besides ordinary uniprocessors like the one OPS2 is implemented on. The results indicated that the algorithm would be efficient on parallel computers and on computers that fetch several words on each memory access in order to improve their processor-memory bandwidths. Whether the algorithm would be efficient on a computer with a cache could not be determined.

Chapter 6 considered the question of whether the Rete Match Algorithm could be made significantly more efficient through the use of specialized hardware. The chapter first explained how to speed up the match process by eliminating most of the references to primary memory and by reducing the amount of processing performed by the nodes. References to primary memory were to be reduced by making the nodes smaller (a possible format was shown) and by storing the data elements in a form that allowed subelements to be accessed in a single memory reference. Processing requirements were to be reduced by storing explicitly the information which the nodes test. These changes would allow production systems to be executed about one order of magnitude faster than the speed of current interpreters. Chapter 6 next considered whether special hardware would improve the efficiency further. It concluded that large increases in speed would result from the use of a very small amount of special hardware plus a few hundred words of microcode.

Calculations were made which showed that a machine with this special hardware would run production systems about two and one-half orders of magnitude faster than current interpreters. Microcode alone would provide a speed increase of about two orders of magnitude. This speed increase would be accompanied by a reduction in the amount of space required to store the productions. When LHSs are compiled into the representation described in this chapter, they decrease in size by a factor of about two.

7.2 Future Research

The obvious continuation of this research is to develop other ways to reduce the cost of running production systems. Several possibilities exist for achieving further reductions, including:

- Compiling the RHSs. This would result in a substantial reduction in the space required to store the productions. But more important, if the match is made two orders of magnitude faster and this is not done, the act phase of the cycle will become the dominant factor in the time cost. Fortunately, compiling the RHSs would be quite easy; almost no research is needed here.

- Use of secondary memory. This could significantly reduce the amount of primary memory used to store the production system. Much research may be required, however; it is not at all clear now how the productions could be paged without slowing the execution excessively. Perhaps the interpreter could extract predictive information from the explicit links between nodes and the implicit links between &P nodes and RHSs.

- Language design. More work in language design might bring about greater efficiency in all areas. It might be possible to design new production system languages that are easier to interpret, that require fewer production firings to perform typical tasks, or that require fewer productions in the system, thus reducing the cost of storing the system.

Another possible continuation of this research is to work on applying the techniques to other languages. Performing pattern matching is often a major part of the cost of running a large Artificial Intelligence program. While the methods developed here might not be as effective as they are for production system interpreters, they might still provide substantial cost reductions in many cases.

I. Comparing Production Systems to Other Programs

The following is an attempt to determine (1) how many lines of a conventional high level language like Lisp or Sail[1] are required to equal one production and (2) how many RHS actions must be performed each second if a production system is to equal the speed of a Lisp or Sail program. The data on which the calculations are based come from Rychener [47, 48] and McCracken [31]. Both Rychener and McCracken used a PDP-10 model KA (a "KA10") as the performance standard. Their original figures are used here, but the final results are converted to use a model KL as the standard. A KL10 is about 4.2 times faster than a KA10; a KL10 executes about 1.3 million instructions per second when running Lisp.

Two comments about the methods employed are in order. First, even though Rychener and McCracken recoded a total of seven programs, only four are listed here. The other three programs were omitted from the calculations because the production system versions of these programs differed too greatly from the original versions. Second, the calculations are probably unfair to the production systems. When it was necessary to convert a range of numbers to a single number, the number that was least favorable to the production system was chosen. The reason for doing so was to insure that if performance standards for production system interpreters were based on these numbers, they would be stringent standards.

Rychener found that a production system version of GPS [36, 18] ran about 20 times slower than a Lisp version of the same program. The production system, which was called GPSR, required from 95 to 185 milliseconds to perform one working memory change. Thus to equal the performance of of the Lisp version, the production system would have to perform a working memory change in about

95 / 20 = 4.8 milliseconds.

To equal the speed of Lisp on a KL, the production system would have to perform an action in

4.8 / 4.2 = 1.1 milliseconds.

The production system contained 206 productions, and the Lisp program contained 616 lines of code. Thus each production is the equivalent of about 3 lines of Lisp:

616 / 206 = 3.0

---

[1] For a description of Sail, see [56] For a description of Lisp, see [60, 53].

These figures are somewhat misleading as the production system had more general abilities than the Lisp program.

In a comparison of a production system version of STUDENT [6, 7] and the original program, Rychener found that the production system, which performed an RHS action in about 160 milliseconds, was about 20 to 30 times slower than the Lisp program. To equal the performance of the Lisp program, therefore, the production system would have to perform an action in

160 / 30 = 5.3 milliseconds.

To equal the performance of Lisp on a KL10, the production system would have to perform an action in

5.3 / 4.2 = 1.3 milliseconds.

The production system contained about 260 productions. The original program contained about 290 Meteor rules.[1]

290 / 260 = 1.1

Rychener had a production system version of a part of the SHRDLU system [61], but the timing information from that production system cannot be used here. The times given for this production system all include the run time of another production system. The size of the system can be used, however. The production system, WBLOX, required 130 productions to perform essentially the same function as 905 lines of Lisp and Micro-planner[2] in SHRDLU.

905 / 130 = 7.0.

McCracken found that HSP, his production system version of Hearsay-II [17, 30], ran 255 times slower than the original Sail version. A run that required 3.6 seconds in Sail required 917 seconds in the production system. A total of 380 working memory changes were made during this run, giving an average of 2.41 seconds per change. The performance of the Sail version could thus be equaled by a production system that performed actions in

2410 / 255 = 9.5 milliseconds.

---

[1] Meteor was an extension to Lisp; it was inspired by the string processing language Comit [62].

[2] For a description of Micro-planner, see [55, 5].

To equal the performance of the SAIL program on a KL10, then, the production system would have to perform an action in

9.5 / 4.2 = 2.3 milliseconds.

In comparing the two programs, McCracken stated that the 310 SAIL statements in the POM subprogram translated into 48 productions.

310 / 48 = 6.5.

The results of this appendix are summarized in the following table.

System	Production Equivalent	KL10 Equivalent Time for Action
GPSR	>3 lines of Lisp	>1 msec.
STUDNT	1 Meteor rule	1 msec.
WBLOX	7 lines of Lisp and Micro-planner	n. a.
HSP	7 lines of Sail	2 msec.

Table I.1. Production Systems vs. Conventional Programs

II. Generality of OPS2 Data Elements

The purpose of this appendix is to demonstrate the generality of OPS2's data elements (and thus to show indirectly the generality of the mechanisms used in OPS2's implementation). Several examples are given here to show that purely mechanical transformations can be applied to other representations to convert them into the OPS2 representation. Most of the representations used in Artificial Intelligence programs belong to one of two classes, sets (sometimes generalized in some way) and colored directed graphs. These two classes include all the representations used in the pure production systems. A few representations from each class have been chosen, and the transformations that might be applied to the representations described below.

To represent a set like

```
{A 17 orange}
```

one might choose a name for the set and store each member of the set in a separate data element.

```
(Set1 member A)
(Set1 member 17)
(Set1 member orange)
```

A common generalization of the set is the "bag", which allows multiple occurrences of objects. The contents of a bag, like those of a set, are unordered.

```
{A 17 orange 17}
```

A bag can be represented in OPS2 by constructing a separate data element for each distinct object. The data element can contain a count of the number of times the object appears in the set.

```
(Bag1 member A 1)
(Bag1 member 17 2)
(Bag1 member orange 1)
```

The simplest colored directed graph is one that uses only one color for the edges. A common example is a string, which is a collection of characters with a "next" relationship between pairs of characters. The string

```
OTTF
```

is

This string could be stored in OPS2 in several ways. If operations on the individual characters were to be performed, one would probably want to store each character in a separate data element. This would require creating a name for the string and a name for each character in the string.

```
(Head String1 G0276)
(G0276 O (NEXT G0277))
(G0277 T (NEXT G0278))
(G0278 T (NEXT G0279))
(G0279 F (NEXT))
```

Name-attribute-value systems also belong to the class of colored directed graphs. These systems contain a few nodes (objects) with many edges going out from them. The colors of the edges are the attributes, and the nodes at the ends of the edges are the values. Often the node from which the edges leave is unnamed because the object is known only by its description. The following, for example, is adapted from Anderson and Gillogly [2].

Type	Site
ID	Rand-11
Operating-system-name	UNIX
Machine-type	PDP-11/45
Guest-account-name	Netguest
Guest-account-password	Netguest
Known-user-set	(JJG RHA RSG)

This describes some unnamed site on the ARPA net. One way to represent it in OPS2 is to choose a name for the site and to put each name-attribute-value triple into a separate data element.

```
(Host199 Type Site)
(Host199 ID Rand-11)
(Host199 Operating-system-name UNIX)
(Host199 Machine-type PDP-11/45)
(Host199 Guest-account-name Netguest)
(Host199 Guest-account-password Netguest)
(Host199 Known-user-set (JJG RHA RSG))
```

Another way to represent this information in OPS2 is in a single list. If one knew before writing the production system precisely how many attributes would be needed to describe sites, one could number the attributes and use lists with that many elements. The first position in the list would hold the value of the first attribute, the second position would hold the value of the second attribute, and so on. The above site could then be represented by

```
(Site Rand-11 UNIX PDP-11/45 Netguest Netguest (JJG RHA RSG))
```

III. Assembly Language Version of &ATOM

The following is an assembly language version of the part of the OPS2 interpreter that would be needed to interpret &ATOM nodes. The first set of instructions below fetches and decodes nodes. The second set executes &ATOM nodes.

```
; Fetch the node and decode it

MAIN:   POP RPEND, RADDR              ; Pop the next address
INTER:  MOVE RNODE,(RADDR)            ; Get the node
        MOVE RTEMP, RNODE             ; Begin to get type field
        ANDI RTEMP, ↑017
        JRST TABLE(RTEMP)             ; Decode the type field

TABLE:  JRST STACKEMPTY               ; TYPE = 0
        JRST ATOM                     ; TYPE = 1
        JRST LEN                      ; TYPE = 2
        JRST LEN+                     ; TYPE = 3
        . . .
        JRST RESERVED                 ; TYPE = 15 decimal

; Node program &ATOM

ATOM:   LDB RTEMP, PTRBRO             ; Extract brother field
        JUMPE RTEMP, ATMNB            ; No brother
        ADD RTEMP, RADDR              ; Use field as offset
        PUSH RSTACK, RTEMP            ; Push the address
ATMNB:  LDB RTEMP, PTRCONST           ; Extract constant field
        IORI RTEMP, ATOMTYPE          ; Add type bits for atom
        LDB RINDEX, PTRINDEX          ; Extract the index field
        CAME RTEMP, ELEM(RINDEX)      ; Make the test
        JRST MAIN                     ; Failure
        AOJA RADDR, INTER             ; Advance address to son
                                      ; & jump into interpreter
```

The following is the program that was shown on the previous page. The instructions are marked with the time (in microseconds) that they would take on a KA10 processor. The instructions without time marks are usually skipped when an &ATOM node is executed. The times assume that the processor has one of the faster available memories, one with a 1.0 microsecond cycle time and a 0.61 microsecond access time. The total time required is 45.51 microseconds. The hypothetical machine from Chapter 6 is about 4.2 times faster than a KA10; thus on that machine, the total time would be 10.8 microseconds.

```
; Fetch the node and decode it

MAIN:   POP RPEND, RADDR                    (3.33 usec)
INTER:  MOVE RNODE,(RADDR)                  (2.87 usec)
        MOVE RTEMP, RNODE                   (1.73 usec)
        ANDI RTEMP, ↑O17                    (1.73 usec)
        JRST TABLE(RTEMP)                   (1.87 usec)

TABLE:  JRST STACKEMPTY
        JRST ATOM                           (1.59 usec)
        JRST LEN
        JRST LEN+
        . . .
        JRST RESERVED


; Node program &ATOM

ATOM:   LDB RTEMP, PTRBRO                   (11.22 usec)
        JUMPE RTEMP, ATMNB                  (1.91 usec)
        ADD RTEMP, RADDR
        PUSH RSTACK, RTEMP
ATMNB:  LDB RTEMP, PTRCONST                 (7.62 usec)
        IORI RTEMP, ATOMTYPE                (1.73 usec)
        LDB RINDEX, PTRINDEX                (5.22 usec)
        CAME RTEMP, ELEM(RINDEX)            (3.10 usec)
        JRST MAIN                           (1.59 usec)
        AOJA RADDR, INTER
```

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

2. J. R. Anderson. *Language, Memory, and Thought.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1976.

3. R. H. Anderson and J. J. Gillogly. *Rand Intelligent Terminal Agent (RITA): Design Philosophy.* Technical report, The Rand Corporation, 1976.

4. R. H. Anderson, M. Gallegos, J. J. Gillogly, R. B. Greenberg, and R. Villanueva. *RITA Reference Manual.* Technical report, The Rand Corporation, 1977.

5. B. G. Baumgart. *Micro-planner Alternate Reference Manual.* Technical report, Stanford Artificial Intelligence Laboratory, 1972.

6. D. G. Bobrow. A Question-answering System for High-school Algebra Word Problems. Proceedings of the Fall Joint Computer Conference, Montvale, NJ, 1964, pp. 591-614.

7. D. G. Bobrow. Natural Language Input for a Computer Problem-solving System. In M. Minsky, Ed., *Semantic Information Processing*, MIT Press, Cambridge, MA, 1968, pp. 146-226.

8. D. G. Bobrow and B. Raphael. New Programming Languages for Artificial Intelligence Research. *ACM Computing Surveys* 6, 3 (September 1974), 153-174.

9. D. G. Bobrow and T. Winograd. An Overview of KRL, A Knowledge Representation Language. *Cognitive Science 1* (January 1977), 3-46.

10. D. G. Bobrow, T. Winograd, and the KRL Research Group. Experience with KRL-0: One Cycle of a Knowledge Representation Language. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 213-222. Cambridge, MA.

11. R. Brooks. Production Systems as Control Structures for Programming Languages. *SIGART Newsletter 63* (June 1977), 33-37.

12. R. Davis, B. G. Buchanan, and E. Shortliffe. Production Rules as a Representation for a Knowledge-based Consultation Program. *Artificial Intelligence 8* (February 1977), 15-45.

13. R. Davis. *Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases.* Technical report, Stanford University, 1976.

14. R. Davis and J. King. An Overview of Production Systems. In E.W. Elcock and D. Michie, Ed., *Machine Intelligence*, Wiley, New York, 1976, pp. 300-332.

15. Digital Equipment Corporation. *DEC System10 Assembly Language Handbook.* Maynard, MA, 1972.

16. Digital Equipment Corporation. *PDP-11/40 Processor Handbook.* Maynard, MA, 1975.

17. L. D. Erman and V. R. Lesser. A Multi-level Organization for Problem Solving Using Many Diverse Cooperating Sources of Knowledge. Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 1975, pp. 483-490. Tbilisi, USSR.

18. G. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, 1969.

19. E. A. Feigenbaum, B. G. Buchanan, and J. Lederberg. On Generality and Problem Solving: A Case Study Using the DENDRAL Program. In B. Meltzer and D. Michie, Ed., *Machine Intelligence*, American Elsevier, New York, 1971, pp. 165-190.

20. C. Forgy. A Network Match Routine for Production Systems. 1974.

21. C. Forgy and J. McDermott. OPS, A Domain-independent Production System. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 933-939. Cambridge, MA.

22. C. Forgy and J. McDermott. *The OPS2 Reference Manual.* Technical report, Department of Computer Science, Carnegie-Mellon University, 1978.

23. S. H. Fuller. Price/performance Comparison of C.mmp and the PDP-10. IEEE/ACM Symposium on Computer Architecture, IEEE Computer Society, 1976, pp. 195-202.

24. F. Hayes-Roth and D. J. Mostow. An Automatically Compilable Recognition Network for Structured Patterns. Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 1975, pp. 246-251. Tbilisi, USSR.

25. F. Hayes-Roth, D. A. Waterman, and D. Lenat. Principles of Pattern-directed Inference Systems. In D. A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 577-601.

26. C. E. Hewitt. How to Use What You Know. Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 1975, pp. 189-198. Tbilisi, USSR.

27. C. E. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence 8* (June 1977), 323-364.

28. D. E. Knuth. *Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

29. D. Lenat. *An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* Ph.D. Th., Stanford University, July 1976.

30. V. R. Lesser, R. D. Fennell, L. D. Erman, and D. R. Reddy. Organization of the Hearsay II Speech Understanding System. *IEEE Transactions on Acoustics, Speech, and Signal Processing ASSP-23*, 1 (February 1975), 11-33.

31. D. McCracken. *A Production System Version of the Hearsay-II Speech Understanding System.* Ph.D. Th., Carnegie-Mellon University, April 1978.

32. D. McDermott and G. Sussman. *The Conniver Reference Manual.* Technical report, MIT AI Lab, 1974.

33. J. McDermott, A. Newell, and J. Moore. The Efficiency of Certain Production System Implementations. In D.A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 155-176.

34. J. McDermott and C. Forgy. Production System Conflict Resolution Strategies. In D.A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 177-199.

35. J. McDermott and J. H. Larkin. *Re-representing Textbook Physics Problems.* Technical report, Department of Computer Science, Carnegie-Mellon University and Physics Department, University of California, Berkely, 1978.

36. A. Newell and H. A. Simon. GPS, A Program that Simulates Human Thought. In E. A. Feigenbaum and J. Feldman, Ed., *Computers and Thought*, McGraw-Hill, New York, 1963, pp. 279-293.

37. A. Newell. Production Systems: Models of Control Structures. In W.G. Chase, Ed., *Visual Information Processing*, Academic Press, New York, 1973, pp. 463-526.

38. A. Newell and J. McDermott. *PSG Manual.* Technical report, Department of Computer Science, Carnegie-Mellon University, 1975.

39. A. Newell. Knowledge Representation Aspects of Production Systems. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 987-988. Cambridge, MA.

40. Mario Pei. *The Story of Language.* Mentor, New York, 1966.

41. L. H. Quam. *Stanford Lisp 1.6 Manual.* Technical report, Stanford Artificial Intelligence Laboratory, 1969.

42. R. Reboh and E. A. Sacerdoti. *A Preliminary QLISP Manual.* Technical report, Artificial Intelligence Center, Stanford Research Institute, 1973.

43. J. R. Rhyne. *On Finding Conflict Sets in Production Systems.* Technical report, Department of Computer Science, University of Houston, 1977.

44. C. Rieger. Spontaneous Computation and its Roles in AI Modeling. In D. A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 69-97.

45. J. F. Rulifson, J. A. Derksen, and R. J. Waldinger. *QA4: A Procedural Calculus for Intuitive Reasoning.* Technical report, Stanford Research Institute, 1972.

46. M. Rychener, C. Forgy, P. Langley, J. McDermott, A. Newell, K. Ramakrishna. Problems in Building an Instructable Production System. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1977, pp. 337. Cambridge, MA.

47. M. D. Rychener. *The STUDNT Production System, A Study of Encoding Knowledge in Production Systems.* Technical report, Department of Computer Science, Carnegie-Mellon University, 1975.

48. M. D. Rychener. *Production Systems as a Programming Language for Artificial Intelligence Applications.* Ph.D. Th., Carnegie-Mellon University, December 1976.

49. M. D. Rychener and A. Newell. An Instructable Production System: Basic Design Issues. In D. A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems*, Academic Press, New York, 1978, pp. 135-153.

50. E. D. Sacerdoti, R. E. Fikes, R. Reboh, D. Sagalowicz, R. Waldinger, and B. M. Wilber.

QLISP -- A Language for Interactive Development of Complex Systems. Proceedings of the National Computer Conference, Montvale, NJ, 1976, pp. 349-356.

51. O. G. Selfridge. Pandemonium: A Paradigm for Learning. In L. Uhr, Ed., *Pattern Recognition*, Wiley, New York, 1958, pp. 237-250.

52. E. H. Shortliffe. *Computer-based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.

53. L. Siklossy. *Let's Talk Lisp.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

54. S. Sternberg. Memory Scanning: New Findings and Current Controversies. *Quarterly Journal of Experimental Psychology 27* (January 1975), 1-32.

55. G. J. Sussman and T. Winograd. *Micro-planner Reference Manual.* Technical report, MIT Project MAC, 1970.

56. K. A. VanLehn. *SAIL User Manual.* Technical report, Stanford Artificial Intelligence Laboratory, 1973.

57. S. A. Vere. Relational Production Systems. *Artificial Intelligence 8* (February 1977), 47-68.

58. D. A. Waterman and A. Newell. PAS-II: An Interactive Task-free Version of an Automatic Protocol Analysis System. *IEEE Transactions on Computers C-25* (April 1976), 402-413.

59. D. A. Waterman and F. Hayes-Roth. An Overview of Pattern-directed Inference Systems. In D. A. Waterman and F. Hayes-Roth, Ed., *Pattern-Directed Inference Systems,* Academic Press, New York, 1978, pp. 3-22.

60. C. Weissman. *Lisp 1.5 Primer.* Dickenson, Belmont, CA, 1967.

61. T. Winograd. *Understanding Natural Language.* Academic Press, New York, 1972.

62. V. H. Yngve. *Comit Programmers' Reference Manual.* MIT Press, Cambridge, MA, 1961.