W&M ScholarWorks

Dissertations, Theses, and Masters Projects | Theses, Dissertations, & Master Projects

1989

Adaptation of LR parsing to production system interpretation

Louis Paul Slothouber
College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation
Slothouber, Louis Paul, "Adaptation of LR parsing to production system interpretation" (1989). *Dissertations, Theses, and Masters Projects.* Paper 1539623785.
https://dx.doi.org/doi:10.21220/s2-xf09-8x17

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9012622

Adaptation of LR parsing to production system interpretation

Slothouber, Louis Paul, Ph.D.

The College of William and Mary, 1989

Copyright ©1989 by Slothouber, Louis Paul. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Adaptation of LR Parsing to Production System Interpretation

A Dissertation
Presented to
The Faculty of the Department of Computer Science
The College of William and Mary in Virginia

In Partial Fulfillment
Of the Requirements for the Degree of
Doctor of Philosophy

---

by
Louis P. Slothouber
August 1989

Copyright © 1989 by Louis Paul Slothouber, All Rights Reserved

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

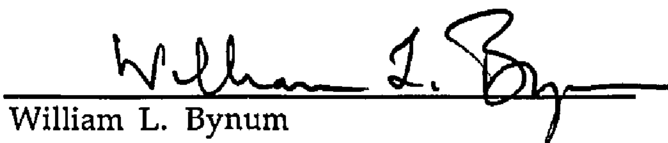
the requirements for the degree of

Doctor of Philosophy

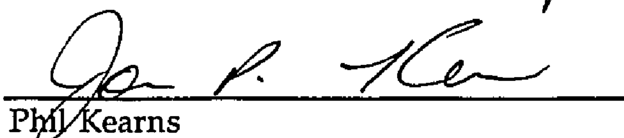
Louis P. Slothouber

Approved, August 1989

W. Robert Collins

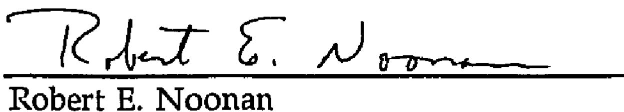


William L. Bynum

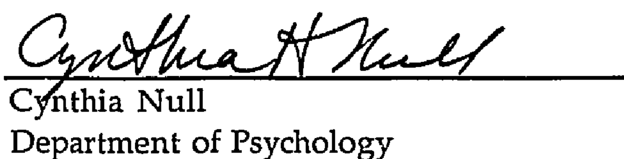


Phil Kearns

Keith Miller



Robert E. Noonan



Cynthia Null
Department of Psychology

For my parents,

who led me to the first page,

and my wife,

who saw me through to the last.

TABLE OF CONTENTS

1. Introduction 1

1.1. Introduction to Production Systems 1

1.1.1. Structure 2

1.1.2. Interpretation 5

1.1.3. Procedural Control 6

1.1.4. Production System Programs, Languages and Architectures 7

1.2. Problems and Related Work 9

1.2.1. The Execution Speed Problem 9

1.2.2. The Procedural Control Problem 12

1.3. Thesis Overview 16

1.3.1. Architectural Requirements of Large Production Systems 16

1.3.2. The Palimpsest Production System Architecture 17

1.3.3. Thesis Organization 19

2. Production Systems 21

2.1. Introduction to Production Systems 21

2.1.1. Post Production Systems 21

2.1.2. Categories and Attributes 23

2.1.3. Additional Predicates 25

2.1.4. Operational Variables and Direct References 26

2.1.5. Negated Patterns 28

2.1.6. Additional Operations 29

2.1.7. Determinism 30

2.2. Production System Definitions and Theorems 31

2.2.1. Structure 31

2.2.1.1. Working Memory 31

2.2.1.2. Production Memory 33
2.2.1.2.1. Conditions 33
2.2.1.2.2. Actions 38
2.2.2. Interpretation 41
2.2.2.1. Backtracking Strategy 41
2.2.2.2. Evaluation Strategy 41
2.2.2.3. Selection Strategy 42
2.2.3. Procedural Control 44
2.2.3.1. Controlled Production Systems 44
2.2.3.2. Examples of Common Control Constructs 48
2.2.3.2.1. Direct Sequencing 48
2.2.3.2.2. Fall-Back Control and Held Result Usage 49
2.2.3.2.3. Selection 49
2.2.3.2.4. Iteration 50
2.2.3.2.5. Modules and Hierarchies 50
2.2.3.2.6. Concurrent Control Constructs 51
2.3. Scope of Controlled Production Systems 52
3. LR Parsing 56
3.1. LR(0) Parsing 57
3.2. Enhancements to LR(0) Parsing 61
3.2.1. CLR(0) Parse Tables 62
3.2.2. Disambiguation Predicates 65
3.2.3. Semantics Function 66
3.2.4. CLR(0) Parsers 67

4. The Palimpsest Parser Production System Architecture 70

4.1. Introduction to Palimpsest Parsers 70

4.1.1. Conflict Resolution of Controlled Production Systems............70

4.1.2. The Fire First Conflict Resolution Function 71

4.1.3. Control Information 72

4.1.4. Memory Support Information...77

4.1.5. Condition Membership Information...79

4.2. Palimpsest Parser Definitions and Theorems 86

4.2.1. Structure 86

4.2.1.1. Working Memory...86

4.2.1.2. Production Memory..88

4.2.2. Interpretation...93

4.2.3. Procedural Control 95

4.3. Scope of Palimpsest Parsers 96

5. Analysis of Palimpsest Parser Performance 102

5.1. Implementation Issues...102

5.1.1. Implementing Working Memory...103

5.1.1.1. A Working Memory Data Structure...................................103

5.1.1.2. Adding Elements to Working Memory.............................104

5.1.1.3. Removing Elements from Working Memory.................107

5.1.2. Implementing Disambiguation Functions 108

5.2. Palimpsest Parser Time Costs...111

5.2.1. Overview of Palimpsest Parser Execution..................................111

5.2.1.1. Finding an Instantiation 112

5.2.1.2. Firing an Instantiation..113

5.2.1.3. Applying Control Rules 113

5.2.2. Effects of Production Memory Size on Time.............................114

5.2.2.1. Worst Case Effect of Production Memory Size on Time114
5.2.2.2. Best Case Effect of Production Memory Size on Time....117
5.2.2.3. Expected Effect of Production Memory Size on Time.....118
5.2.3. Effects of Working Memory Size on Time....120
5.2.3.1. Worst Case Effect of Working Memory Size on Time...120
5.2.3.2. Best Case Effect of Working Memory Size on Time.......121
5.2.3.3. Expected Effect of Working Memory Size on Time........121
5.3. Palimpsest Parser Space Costs....121
5.3.1. Overview of Palimpsest Parser Composition....121
5.3.1. Effects of Production Memory Size on Space....122
5.3.2. Effects of Working Memory Size on Space....123
5.4. Empirical Tests of Palimpsest Parsers....123
5.4.1. Description of The Testing Methodology....124
5.4.2. Empirical Results....125

6. Palimpsest Parser Enhancements 129

6.1. Additional Features....129
6.1.1. Backward-Chaining Evaluation....129
6.1.2. Separate Compilation Units....132
6.1.3. Conventional Conflict Resolution....132
6.2. Future Research....134
6.2.1. Modular Production System Language....134
6.2.2. Performance Optimizations....134
6.2.3. Explanatory Capability....134
6.2.4. Uncertainty....135
6.2.5. Reasoning About Controlled Production Systems....135
6.2.6. Concurrency....136

7. Conclusion **137**

Bibliography **139**

Appendix A. Notation **144**

Appendix B. A Complete Production System Example **147**

B.1. Designing a Production System 147

B.2. Applying The Palimpsest Transformation 153

The Palimpsest Grammar for LOADBAY 156

B.3. Execution of the LOADBAY Palimpsest Parser 160

ACKNOWLEDGEMENTS

I wish to express my deepest appreciation to my advisor, Bob Collins, for his patient guidance, and criticism throughout this investigation. I am also indebted to Randall Meyer for his insightful comments regarding early portions of this work, and to the members of my committee for their careful reading and criticism of this manuscript. Laurie King and Stef Lucas deserve special recognition for their words of encouragement, and for their much needed last minute aid. Most importantly, I wish to thank my wife, Karen, for her invaluable support: emotional, financial, and grammatical.

LIST OF TABLES

No. Name Page

5.1 Worst Case Results....127
5.2 Best Case Results....128
5.3 Typical Production System Results....130

LIST OF FIGURES

No.	Name	Page
1.1	A Sample Working Memory	2
1.2	A Sample Production	3
1.3	A Sample Instantiation	5
1.4	Unordered Productions	8
1.5	Sequential Productions	8
1.6	Simple Implementation of the Match Phase	10
2.1	Direct References vs. Operational Variables	27
2.2	A Sample Production	30
2.3	Sets Containing Working Memory Elements	45
3.1	A Canonical Set of LR(0) Parse Tables	59
3.2	Canonical CLR(0) Parse Tables	64
3.3	Alternate Form of CLR(0) Parse Tables	65
4.1	A Controlled Production System	73
4.2	CLR(0) Tables for FIGURE 4.1	74
4.3	New CLR(0) Parse Tables for FIGURE 4.1	82
4.4	CLR(0) Parse Tables for Turing Machines	101
5.1	A Working Memory Data Structure	103
5.2	An Ada Implementation of Function *d*	108
5.3	Testing Many Element Predicates in Function *d*	109
5.4	Procedures Called by Function *d*	110
5.5	A Diagram of Palimpsest Parser Execution	112
5.6	Worst Case Effect of **P** on Time	127
5.7.	Best Case Effect of **P** on Time	128
5.8	Typical Effect of **P** on Time	130
6.1	A Backward Chaining Controlled Production System	133
B.1	An Initial *LOADBAY* Working Memory	150
B.2.	The *LOADBAY* Control Grammar	152
B.3.	The *LOADBAY* Productions (part 1)	153
B.3.	The *LOADBAY* Productions (part 2)	154
B.4.	The *LOADBAY* Element Class Table	155
B.5	The *LOADBAY* Classification Procedure	156

ABSTRACT

This thesis presents such a new production system architecture, called a *palimpsest parser,* that adapts LR parsing technology to the process of controlled production system interpretation. Two unique characteristics of this architecture facilitate the construction and execution of large production systems: the rate at which productions fire is independent of production system size, and the the modularity inherent in production systems is preserved and enhanced. In addition, individual productions may be evaluated in either a forward or backward direction, production systems can be integrated with other production systems and procedural programs, and production system modules can be compiled into libraries and used by other production systems.

Controlled production systems are compiled into palimpsest parsers as follows. Initially, the *palimpsest transformation* is applied to all productions to transform them into context-free grammar rules with associated disambiguation predicates and semantics. This grammar and the control grammar are then concatenated and compiled into modified LR(0) parse tables using conventional parser generation techniques. The resulting parse tables, disambiguation predicates, and semantics, in conjunction with a modified LR(0) parsing algorithm, constitute a palimpsest parser. When executed, this palimpsest parser correctly interprets the original controlled production system. Moreover, on any given cycle, the palimpsest parser only attempts to instantiate those productions that are allowed to fire by the control language grammar. Tests conducted with simulated production systems have consistently exhibited firing rates in excess of 1000 productions per second on a conventional minicomputer.

Adaptation of LR Parsing to Production System Interpretation

1. Introduction

In recent years production systems have proven to be valuable artificial intelligence tools for representing and applying domain specific knowledge; yet, many characteristics of current production system architectures make large production systems impractical. For example, the rate of production firing is dependent upon production system size; big production systems execute slower than smaller ones. Also, specifying and imposing procedural control via conflict resolution destroys the modularity inherent in production systems. This thesis presents a new production system architecture, based upon the theory of LR parsing, that facilitates the construction and execution of large production systems.

1.1. Introduction to Production Systems

The production system was introduced in 1943 by Post[1] as a general computational mechanism equivalent to Universal Turing Machines[2]. Since then the production system has been adapted and applied to numerous problems throughout computer science. Markov algorithms[3], the Floyd-Evans production language[4] and modern context-free grammars[5] are all adaptations of Post production systems. More recently, production systems have been adapted to the task of representing and applying domain specific knowledge in a manner which models human cognitive processes, so-called *rule-based expert systems*. Such production systems bear little resemblance to Post production systems. This section presents a brief introduction to these modern production systems and related terminology.

1 Post, 1943.
2 Aho and Ullman, 1972, p. 29.
3 Minsky, 1967, CHAPTER 12.
4 Floyd, 1961.
5 Aho and Ullman, 1972.

```
(food     type:meat     loc:doghouse)
(vehicle  type:truck    loc:I95)
(animal   name:pudwuji  species:dog  loc:doghouse  hungry:true)
(person   name:louis    age:27       loc:doghouse  sex:M)
(person   name:karen    age:26       loc:house     sex:F)
(concept  name:joy      type:emotion)
(concept  name:blue     type:color)
```

FIGURE 1.1. A Sample Working Memory

1.1.1. Structure

A production system is composed of a finite set of structures called *productions* and a set of data elements called *working memory.* Each data element, called a *working memory element,* is a string of symbols to which some meaning is usually ascribed. Most frequently, working memory elements are understood to represent objects[6] where each consists of values assigned to the *attributes* of that object. All working memory elements that describe objects of the same type or *category* have the same attributes[7]. For example, the working memory elements described in FIGURE 1.1 represent seven objects: a food item, a truck, a dog, two people, and two concepts. Each working memory element is denoted by a category followed by one or more attributes (boldface) with their associated values.

A production is composed of a *name,* a *condition* that tests the state of working memory, the symbol ⇒, and an *action* that modifies the state of working memory. For example, the sample production of FIGURE 1.2, called *feed-a-dog,* contains the condition:

```
    (food   type=meat)
    (animal species=dog  loc=#1.loc  hungry=true)
NOT (animal size>#2.size loc=#1.loc  hungry=true)
```

6 The term *object* is used here to represent a person or thing to which action or thought is directed.

7 In the literature, the term *element class* is used in place of *category.* In this thesis, an element class is a subset of a category (cf. SECTION 4.2).

```
feed-a-dog:
    (food   type=meat)
    (animal species=dog  loc=#1.loc  hungry=true)
NOT (animal size>#2.size loc=#1.loc  hungry=true)
⇒
CHG (#2,    hungry:false)
ADD (object type:bone    loc:#1.loc)
REM (#1)
```

FIGURE 1.2. A Sample Production

and the action:

```
CHG (#2,    hungry:false)
ADD (object type:bone    loc:#1.loc)
REM (#1)
```

Every condition contains one or more *patterns*, such as:

```
(animal species=dog  loc=#1.loc  hungry=true)
```

that defines a subset of some category; in this case *animal*. For example, this pattern defines a set that contains the element:

```
(animal  name:pudwuji species:dog loc:doghouse hungry:true)
```

Patterns preceded by *NOT* are said to be *negative;* all other patterns are *positive*. A pattern may make *direct references* to attributes of a working memory element that matches a positive pattern within the same condition (e.g., *#1.loc* references the *loc* attribute of a working memory element matching the first pattern in the condition)[8]. A condition is *satisfied* when

1. working memory contains at least one element of every set defined by the positive patterns in the condition, and
2. working memory contains no elements of any set defined by the negative patterns in the condition.

The sample working memory of FIGURE 1.1 satisfies the condition of the sample production in FIGURE 1.2, where the working memory elements

8 For reasons beyond the scope of this introduction, the literature uses operational variables, not direct references, to reference attribute values of other patterns (cf. SECTION 2.1.4).

```
(food     type:meat      loc:doghouse)
(animal   name:pudwuji species:dog loc:doghouse hungry:true)
```

are said to *match* the positive patterns

```
(food    type=meat)
(animal species=dog  loc=#1.loc  hungry=true)
```

respectively.

Every action consists of one or more *operations* that may be applied to working memory if the corresponding condition is satisfied. These operations may also contain direct references to the attributes of positive patterns in that condition. For example, the operation

```
ADD (object type:bone     loc:#1.loc)
```

adds a new element to working memory,

```
REM (#1)
```

removes the working memory element that matches the first pattern in the condition, and

```
CHG (#2, hungry:false)
```

modifies the ***hungry*** attribute of the working memory element that matches the second pattern in the condition.

The sample production *feed-a-dog* may be read as follows:

If working memory contains:
- *a piece of meat, and*
- *a hungry dog at the same location as the meat,*

And if working memory does not contain:
- *a bigger, hungry animal at the same location as the meat,*

Then:
- *assert that the dog is no longer hungry,*
- *add a new bone to working memory, and*
- *remove the meat from working memory.*

```
⟨[feed-a-dog:
      (food   type=meat)
      (animal species=dog   loc=#1.loc   hungry=true)
  NOT (animal size>#2.size  loc=#1.loc   hungry=true)
⇒
  CHG (#2, hungry:false)
  ADD (object type:bone  loc:#1.loc)
  REM (#1) ],
 [(food    type:meat    loc:doghouse),
  (animal  name:pudwuji  species:dog  loc:doghouse  hungry:true),
  - ] ⟩
```

FIGURE 1.3. A Sample Instantiation[9]

1.1.2. Interpretation

An *instantiated production,* also called an *instantiation,* consists of a production and a list of working memory elements that match the patterns in that production's condition. It is assumed that no elements of working memory match the negative patterns in the condition. For the sample production of FIGURE 1.2 and working memory of FIGURE 1.1, an instantiation is shown in FIGURE 1.3.

To *fire* an instantiation[10]:

1. All direct references in the action of the production are replaced with the corresponding elements and attribute values from the working memory elements of the instantiation.
2. The operations of the production's action are performed sequentially on working memory.

For the sample instantiation of FIGURE 1.3, step (1) would instantiate the direct references in the production's action to

---

9 The – is used in place of the third working memory element because the corresponding third pattern is negative, and matches no working memory elements.

10 Throughout the literature, the term *fire* is often applied to productions (i.e., instantiated productions), in which case a set of matching working memory elements is assumed to exist.

```
CHG [ (animal  name:pudwuji  species:dog ...), hungry:false]
ADD (object    type:bone     loc:doghouse)
REM [ (food    type:meat     loc:doghouse)]
```

and step (2) would first change the working memory element

```
(animal  name:pudwuji  species:dog  loc:doghouse  hungry:true)
```

into

```
(animal  name:pudwuji  species:dog  loc:doghouse  hungry:false)
```

and then add the element

```
(object  type:bone     loc:doghouse)
```

to working memory, and finally remove the element

```
(food    type:meat     loc:doghouse)
```

from working memory.

The *interpretation* of a production system is commonly defined by the following three phases, called the *recognize-act cycle:*

1. *Match Phase:* Generate the set of current instantiations, called the *conflict set.*
2. *Conflict Resolution Phase:* Choose one instantiation from the conflict set, or halt if the conflict set is empty.
3. *Act Phase:* Fire the chosen instantiation.

These three steps are repeated, in order, until no more instantiations can be found, or a *halt* operation is performed.

1.1.3. Procedural Control

In the procedural languages, such as Pascal or Ada, flow of control information is unambiguously specified. Such *procedural control* is not found in production systems, where each production is supposed to be an independent module. Ideally, the order in which productions are fired depends entirely upon the contents of working memory; no production directly calls another production. This lack of procedural control allows production systems to exhibit two very desirable properties:

1. Production systems can react quickly to small changes in the data without explicitly describing how to search for those changes.
2. Production systems are highly modular[11], a property that enhances their extensibility, flexibility, and programmability.

Unfortunately, most problem domains are not ideal, and some method of controlling the order of production invocation is necessary. For example, the efficient search of a large problem space may require that more promising productions be applied before less promising productions are applied. Many control constructs, such as *sequencing, subroutining,* and *iteration,* are considered essential for most artificial intelligence applications[12].

The conflict resolution phase, by choosing an instantiation to fire on each cycle, determines the order in which productions fire. Most conventional production system architectures take advantage of this property of conflict resolution to impose procedural control[13,14]. A conflict resolution strategy (i.e., an algorithm for choosing an instantiation) is specified along with every production system. The production system can then exploit knowledge about that strategy by creating and referencing new "bookkeeping" working memory elements. For example, suppose it is necessary for production *B* in FIGURE 1.4 to fire immediately after production *A*. If the conflict resolution strategy chooses the instantiation containing the newest working memory element, then productions *A* and *B* can be modified as in FIGURE 1.5 to achieve this ordering.

1.1.4. Production System Programs, Languages and Architectures

There is a clear distinction between programs (e.g., a word processor), programming languages (e.g., Pascal), and runtime architectures (e.g., activation stack and heap). There is a similar distinction between production systems, production system languages, and production system architectures.

---

11 Georgeff, 1982, pp. 178-180.
12 Rychener, 1977.
13 ibid.
14 McDermott and Forgy, 1978.

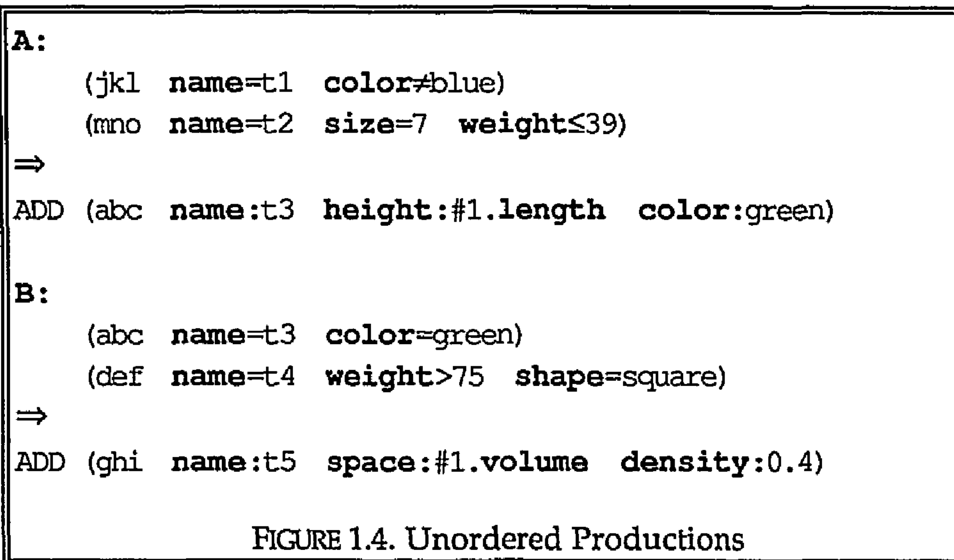


FIGURE 1.4. Unordered Productions

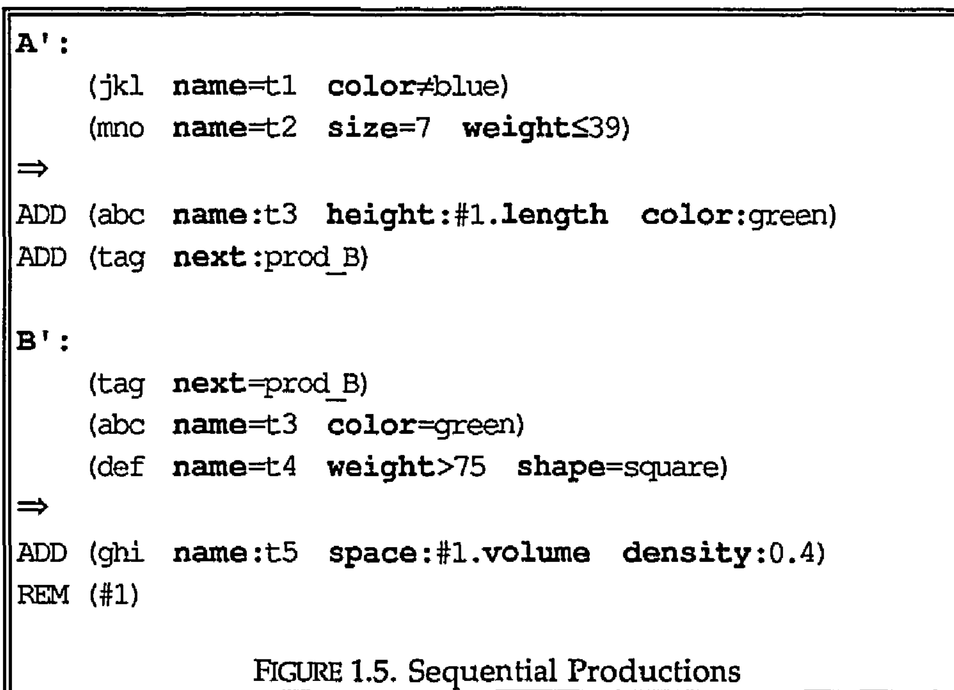


FIGURE 1.5. Sequential Productions

Production systems are programs; they process information, albeit differently than procedural language programs. A *production system language* is a syntax definition with associated semantics in which production system programs are written. A *production system architecture* is a general runtime architecture capable of interpreting production systems. A *production system interpreter* is an implementation of a production system architecture that has

been tailored for use with a specific production system language and target machine.

More detailed introductions to production systems may be found in CHAPTER 2 and elsewhere[15,16].

1.2. Problems and Related Work

Production system architectures have always had two fundamental problems: execution speed is dependent upon production system size, and imposing procedural control on productions is difficult. The ever increasing sizes of real production systems have exacerbated these problems, and motivated much of production system research. This section describes such research, and suggests that existing architectures and control methodologies are inadequate when applied to large production systems.

1.2.1. The Execution Speed Problem

Production systems are slow. In general, large production systems execute more slowly than smaller ones, even if most of their productions are never used[17]. This is quite unlike a procedural programming language such as Pascal or Ada, for which the time required to perform a statement is independent of the total number of statements in the program.

To understand this dependence of speed upon size one must look closely at the process of production system interpretation. Let C_i^+ be the number of positive patterns and C_i^- be the number of negative patterns in a production number *i*, and consider the simple implementation of the match phase in FIGURE 1.6. If a production system contains **P** productions, working memory contains **W** elements, and the average number of positive and negative patterns in all the conditions are C^+ and C^- respectively, then this

15 Davis and King, 1976.
16 Waterman and Hayes-Roth, 1978.
17 Forgy, 1979.

```
Foreach production i:
  Foreach working memory element w1 that may match the 1st positive pattern:
    Foreach working memory element w2 that may match the 2nd positive pattern:
      . . .
      Foreach working memory element wP that may match the Ci+th positive pattern:
        If w1 matches the 1st positive pattern and
           w2 matches the 2nd positive pattern and
                . . .
           wP matches the Ci+th positive pattern Then
              satisfied := true;
        Else
              satisfied := false;
        index := Ci+;
        While (index < Ci+ + Ci-) and satisfied Do
          index := index + 1;
          Foreach working memory element windex that may match the (index - Ci+)th
                negative pattern:
            If windex matches the (index - Ci+)th pattern Then
              satisfied := false;
        Endwhile;
        If satisfied Then
           Add [i, <w1, w2, ..., wC>] to the conflict set.
```

FIGURE 1.6. Simple Implementation of the Match Phase

implementation of the match phase requires at least $P \cdot W^{C^+}$ evaluations[18] of the first, positive *if* test and $P \cdot W \cdot C^-$ evaluations of the second, negative *if* test on every recognize-act cycle.

For large production systems the computation required by the match phase easily dominates the computation of the other two phases[19,20]. Most matching algorithms reduce this computation using *condition membership* and *memory support* information to *filter* out inactive productions and working memory elements. Condition membership associates with each pattern a running count of the number of working memory elements that partially satisfy it; only productions with partially satisfied positive patterns need to be examined on each cycle. Memory support indicates for each pattern the

18 $\Sigma W^{C_i^+} \geq P \cdot W^{(\Sigma C_i^+)/P}$. Hardy, Littlewood, and Pólya, 1959.

19 Forgy, 1982.

20 Rhyne, 1977.

subset of working memory that partially matches it; only this subset of working memory needs to be matched with the pattern. The process of collecting and maintaining such information while working memory is modified is called *indexing*. Various indexing schemes have been implemented[21,22,23] and have demonstrated significant reductions in match phase computation; but even if complete condition membership and memory support information is provided, the match algorithm must still iteratively examine the remaining productions and working memory elements.

The Rete match algorithm[24,25] avoids this extra examination by also maintaining *condition relationship* information; that is, information about which working memory elements satisfy the relationships between patterns within productions. It translates a production system into a network of recognizer nodes, each of which tests a particular attribute of a working memory element. Only those working memory elements that satisfy this test are passed on to further nodes in the network. For each change to working memory input to this network, a set of changes to the current conflict set is output. Because the cost of maintaining condition relationship information is high, it is assumed that only small changes are made to working memory on each cycle.

It is argued however, that the cost of maintaining condition relationship information may exceed the cost of recomputing that information, as needed, on each cycle[26]. The TREAT algorithm is based upon that assumption[27], and implements full condition membership and memory support filters as in Rete, but condition relationship is replaced by *conflict set support* information. That is, the conflict set is explicitly retained between recognize-act cycles, allowing the search for instantiations to be restricted to those that came about due to the most recent working memory modifications. Empirical tests sug-

21 McDermott, Newell, and Moore, 1978.
22 Rhyne, 1977.
23 Rieger, 1978.
24 Forgy, 1979.
25 Forgy, 1982.
26 McDermott, Newell, and Moore, 1978, p. 160.
27 Miranker, 1987.

gest that the TREAT algorithm typically outperforms the Rete algorithm[28]. Other, less complex match algorithms have also outperformed the Rete algorithm in specific problem domains[29].

The best and worst case time complexities of the Rete algorithm with respect to the number of productions P are $O(\log_2 P)$ and $O(P)$, respectively; and, it is expected that most production systems will be $O(P)$[30]. In absolute terms, one of the fastest production system languages to use the Rete algorithm, OPS83[31], might be expected to fire from 25 to 50 productions per second for a large production system with a large working memory, such as R1[32,33]. Newer matching algorithms may be slightly faster by a small constant factor; but even in the best case, execution speed is still dependent upon production system size. This dependency effectively limits the size of practical production systems using these matching algorithms. Conventional wisdom suggests that "substantial further increases [in execution speed] are not likely through software techniques"[34] and most current research in efficient production system interpretation investigates the use of parallel hardware support for production system interpreters[35,36,37].

1.2.2. The Procedural Control Problem

Production systems are naturally free from procedural control; no production can directly call another. Yet, a clear and efficient production system solution in most problem domains requires some procedural control[38]. As the sizes of real production systems have grown, so too has the

28 Miranker, 1987.
29 Nuutila, Kuusela, Tamminen, Veilahti, Arkko, and Bouteldja, 1987.
30 Forgy, 1979, p. 106.
31 Forgy, Gupta, Newell, and Wedig, 1984, p. 117.
32 McDermott, 1980.
33 Forgy, Gupta, Newell, and Wedig, 1984, p. 117.
34 Forgy, Gupta, Newell, and Wedig, 1984, p.116.
35 Forgy, Gupta, Newell, and Wedig, 1984.
36 Forgy, Gupta, Newell, and Wedig, 1986.
37 Stolfo and Shaw, 1982.
38 Rychener, 1979.

need for a control methodology that allows procedural control to be clearly specified and efficiently imposed.

Many production system languages do not allow procedural control to be explicitly specified; the order in which productions are fired is pre-determined by the production system architecture. Such architectures are typically backward-chaining[39] (e.g., MYCIN[40,41], EXPERT[42], and KAS[43]) and backtrack to fire all instantiations in conflict sets. Also, PROLOG[44] can be viewed as a backward-chaining production system language in which procedural control is prescribed by the AND/OR tree defined by the productions.

Most forward-chaining production system architectures support a more flexible procedural control methodology in which bookkeeping working memory elements are used to exploit knowledge of a conflict resolution strategy (e.g., OPS5[45], YAPS[46] and CLIPS[47]). Some systems augment this methodology by filtering out some productions, prior to the match process, that cannot match control-related working memory elements (e.g., GRAPES[48]). An even more flexible control methodology involves the use of meta-level productions or algorithms to dynamically modify the conflict resolution strategy[49,50] (e.g., TEIRESIAS[51], and OPS83[52]).

---

39 Backward-chaining, also called goal-driven evaluation attempts to satisfy a *goal production* by examining the actions of other productions to find one that, if fired, will help to instantiate the goal production.

40 Shortliffe, 1976.

41 Buchanan and Shortliffe, 1984.

42 Weiss and Kulikowski, 1981.

43 Duda, Gaschnig, and Hart, 1979.

44 Clocksin and Mellish, 1982.

45 Brownston, Farrell, Kant, and Martin, 1985.

46 Allen, 1982.

47 Giarratano, 1988.

48 Sauers and Farrell, 1982.

49 Davis and Buchanan, 1977.

50 Davis, 1980.

51 Davis, 1976.

52 Forgy, 1984. Note, OPS83 uses a meta-level procedural program to perform dynamic conflict resolution, not meta-level productions.

An unfortunate side-effect of conflict resolution based control methodologies is the destruction of the modularity inherent in production systems[53]. Because control knowledge is embedded within productions, adding or removing a production requires knowledge of the contents of all other productions. This situation can be alleviated somewhat with the use of working memory elements that specify goals or contexts; only productions applicable in some current context can be applied[54]. Some production system architectures can exploit such production grouping to reduce the computation of the match phase[55]. Also, it is argued that such grouping facilitates the design and maintenance of production systems[56,57].

Although most production system languages neither require nor enforce such grouping, some do provide facilities that allow grouping control information to be specified outside of productions[58]. That is, the assignment of productions to specific groups or rule-sets, and the order in which rule-sets are to be applied, are both specified separately from the productions. Regardless of the grouping method, other control constructs, such as iteration and subroutining, must still be implemented using methodologies based on conflict resolution[59].

Another approach, which is both formal and general, explicitly specifies all procedural control for a production system with a separate *control language*[60], where the primitive symbols of that language are production names. The strings of production names in this language explicitly define the allowable sequences of production firings. On any cycle, the control language defines the set of productions that are allowed to fire. Further, meta-level productions with empty actions are introduced to allow dynamic selection of control paths (i.e., production sequences) from among those specified by the

53 Georgeff, 1982, pp. 178-180.
54 Rychener, 1979.
55 Forgy, 1979, p. 101-102.
56 Jacob and Froscher, 1986.
57 Jacob and Froscher, 1985.
58 Barker and O'Connor, 1989.
59 Rychener, 1979.
60 Georgeff, 1982.

control language. Control information specified in this way can be used to impose procedural control by modifying the recognize-act cycle as follows:

0. Find the subset of productions specified by the control language.
1. Find all instantiations for this subset of productions.
2. Choose an instantiation. If none exists, halt.
3. Fire the chosen instantiation, and return to step (0).

Production systems employing this approach, called *controlled production systems*, allow entire plans, or sequences of productions, to be treated as modules. Also, by separating control information from production information the modularity inherent in production systems is retained[61], and production systems may be more easily constructed, verified, and modified[62].

Although this approach seems well suited to the task of specifying control for large production systems, the more efficient and complex matching algorithms were not designed with controlled production systems in mind. As the modified recognize-act cycle above suggests, a control language could be used to *pre-index* allowable productions for both the simplistic and indexing match algorithms. However, any decrease in the computational cost of the match would be offset by the cost of computing the allowed productions specified by a control language grammar, and the use of a relatively inefficient matching algorithm.

This type of pre-indexing also appears to be impractical with the Rete and TREAT match algorithms. Within Rete and TREAT networks, nodes are often shared by many patterns from many productions. Isolating and de-activating the nodes for an arbitrary set of productions seems prohibitively difficult. While instantiations containing productions that are not specified by the control language could be removed from the conflict set prior to the conflict resolution phase, this additional computation does nothing to decrease the workload of the match phase, and would *increase* the computational cost of interpreting production systems.

---

61 Georgeff, 1982.

62 Kowalski, 1979.

1.3. Thesis Overview

1.3.1. Architectural Requirements of Large Production Systems

Modularity is generally recognized as the only guideline available for mastering the complexity of design and implementation of large and complex programs[63]. The term modularity here implies both *hierarchical structure,* the ability to construct large modules from smaller ones, and *independence,* the ability to understand and implement modules independently of one another. It might seem that production systems, in which each production is an independent module, are ideally suited to programming in the large. However, most existing production system architectures rely, to a greater or lesser extent, upon conflict resolution to provide procedural control, and destroy this production independence. Also, with the exception of controlled production systems, no existing production system architecture supports hierarchically structured modules.

In addition to supporting modularity, large production systems require that an architecture fires productions at a rate that is independent of the total number of productions. Any dependence of speed upon size effectively limits the size of useful production systems. Unfortunately, the speeds of matching algorithms appear to be intrinsically dependent upon the number of productions being matched. One resolution to this apparent conundrum is provided by controlled production systems, in which separate procedural control information can be used to group productions into hierarchically structured modules. In such systems, only a small number of productions are "allowed" to fire on any given cycle, namely, those productions in the currently executing module[64]. Because each module should realize a single and simple conceptual function of the system, module sizes are automatically restricted as a by-product of the design process[65]. By using control information

63 Ghezzi and Jazayeri, 1982, p. 29.

64 In practice, control is often imposed within the modules as well, and the number of productions that are allowed to fire on any cycle is further reduced.

65 Ghezzi and Jazayeri, 1982, p. 29.

to direct and constrain the matching algorithm the rate of production firing is bounded by a constant related to the maximum module size[66].

For large production systems, modularity is the most important issue in the design of production system languages and the architectures that support them. A language enforces modularity so that the complexity of large production systems can be managed. An architecture supports that modularity to efficiently interpret those production systems.

1.3.2. The Palimpsest Production System Architecture

As described above, controlled production systems maintain production modularity and allow the specification of hierarchical production modules. Unfortunately, the faststest production system architectures in use today are incompatible with this control methodology. The goal of this work has been to develop a new production system architecture that uses the control language of a controlled production system to direct the search for instantiations. Such an architecture could support new production system languages that facilitate the design and construction of large production systems.

This thesis presents a new production system architecture, called a *palimpsest parser*[67], that adapts LR parsing technology to the process of controlled production system interpretation. LR parsers were chosen as the basic program structure for three reasons. First, LR parsers have four desirable properties; they are fast, well understood, have a self-introspective capability, and can be automatically generated from context-free grammars[68]. Second, given an LR parser generated from a control language grammar and a string

---

66 Note, this is only true if the architecture actually uses the control information to constrain the search for instantiations. The firing rate of an architecture that searches for all instantiations and then removes those that are not allowed to fire is still dependent upon the total number of productions. The hierarchical structure aspect of modularity is an architectural issue, *and* a language issue.

67 A *palimpsest* is a written document, typically on vellum or parchment, that has been written upon several times, often with the remnants of earlier, imperfectly erased writing still visible. Similarly, a palimpsest parser is a program that has been derived, in stages, from other program representations (i.e., first a controlled production system and then an augmented grammar), with remnants of those earlier program representations still visible.

68 Feyock, 1984.

of the names of productions that have already fired, the set of productions that are allowed to fire next can be easily determined. Third, LR parsers have been successfully used to implement small, backward chaining, propositional[69] expert systems[70].

Controlled production systems are compiled into palimpsest parsers as follows. Initially, the *palimpsest transformation* is applied to all productions to transform them into context-free grammar rules with associated disambiguation predicates and semantics. This grammar and the control grammar are then concatenated and compiled into modified LR(0) parse tables using conventional parser generation techniques. The resulting parse tables, disambiguation predicates, and semantics, in conjunction with a backtracking LR(0) parsing algorithm, constitute a palimpsest parser. When executed, this palimpsest parser correctly interprets the original controlled production system. Moreover, on any given cycle, the palimpsest parser only attempts to instantiate those productions that are allowed to fire by the control language grammar. Tests of simulated production systems[71] have consistently exhibited firing rates in excess of 1000 productions per second on a conventional minicomputer.

Additional features of the palimpsest parser production system architecture include[72,73]:

- Individual productions and production modules may be evaluated in either a forward, backward, or bi-directional manner.
- Separately compiled production system units, similar to Ada packages, may be constructed.
- Palimpsest parsers are self-contained and can be integrated with, procedural language programs.

---

69 The term *propositional* is used here to imply that patterns consist of simple tokens without variables.

70 Collins and Feyock, 1985.

71 Simulated production systems had the following characteristics: 200 productions, 500 working memory elements, an average of 3 patterns and 3 actions per production.

72 Collins and Slothouber, 1988.

73 cf. Section 6.1.2.

This thesis describes a production system architecture, not a production system language, but languages and architectures are related. An architecture essentially defines the semantic capabilities of the languages it supports. In this sense, all languages for a given architecture will be somewhat similar. Presentation of the palimpsest production system architecture is more formal than is usually the case in the literature in order to emphasize the new ideas underlying this architecture rather than any specific implementation or production system language. However, to illustrate these ideas example productions and working memory elements are presented using a simple production system language. This language is arbitrary, and should not be confused with the underlying architecture.

1.3.3. Thesis Organization

CHAPTER 2 begins by contrasting modern production systems with Post production systems. More formal definitions of the structure, interpretation, and control of production systems are then presented. The chapter concludes by proving the equivalence of this production system formalism and deterministic Turing machines. This proof and formal description of production systems is original to this thesis.

Although the theory of LR parsing has been formally defined, many of the features found in modern machine-generated LR parsers have not been integrated into these definitions. CHAPTER 3 extends the definitions of LR(0) parsing to include these new features.

CHAPTER 4 begins with an informal explanation of the adaptation of LR parsing to the process of controlled production system interpreation. Formal definitions of palimpsest parsers and related concepts are then presented. The palimpsest transformation is introduced and proven to be applicable to all controlled production systems. The chapter concludes by proving that for every deterministic Turing machine there exists an equivalent palimpsest parser. The material in this chapter is original to this thesis.

CHAPTER 5 presents the time and space cost calculations for palimpsest parsers. Results of tests on simulated production systems are then presented to support these theoretical costs.

CHAPTER 6 describes, some additional features of palimpsest parsers. Ideas for future research related to palimpsest parsers are presented.

CHAPTER 7 concludes the thesis.

Appendix A defines the notation used throughout the thesis. Some of this notation is non-standard. The reader is advised to peruse this appendix before reading the following chapters. Also, notation defined within a definition is often taken for granted thereafter. When in doubt, refer to Appendix A.

Appendix B presents a complete production system example. First, the problem to be solved is specified. A top-down design of the solution is then presented that uses control constraints to divide the problem into manageable subproblems. The palimpsest transformation is then applied to the resulting controlled production system to produce a palimpsest parser. Finally, a trace of the execution of that palimpsest parser is presented.

2. Production Systems

This chapter provides an introduction to production systems and related concepts. The first section informally describes Post production systems and various additional features found in most conventional production system architectures. The second section presents more formal definitions of the structure, interpretation, and control of production systems. The final section addresses the scope of controlled production systems. In particular, their equivalence to Turing machines is demonstrated.

2.1. Introduction to Production Systems

This section describes Post production systems and various additional features common to most conventional production system architectures. The purpose of this section is to give the reader some understanding of basic production system concepts before formal definitions are presented in SECTION 2.2.

2.1.1. Post Production Systems

A Post production system (PPS) is composed of a set of data elements called *working memory*, and a set of *productions* that modify working memory. Let Σ be a finite set of *primitive symbols*. Elements of Σ^+, the set of non-empty strings of primitive symbols, are called working memory elements[74]. Working memory is a subset of Σ^+. There is also a set Ψ of *variables*, where $\Sigma \cap \Psi$ is empty. Productions are formed from strings in $(\Sigma \cup \Psi)^+$. In particular, a production is of the form

$$a\ b\ \ldots c \text{ produce } d$$

[74] Working memory elements are called *enunciations* by Post.

where $a, b, \ldots, c$, and d are strings in $(\Sigma \cup \Psi)^+$. The strings $a, b, \ldots, c$ are called *patterns*.

Such productions are interpreted as follows. If each pattern $a, b, \ldots, c$ in a production can be pattern matched with an element in working memory, such that variables in the pattern are identified consistently throughout the production with strings in Σ^*, then d, with identical variable replacements, is added to working memory. The interpretation of a PPS applies this process non-deterministically for all productions in the PPS until no more new elements may be added to working memory.

EXAMPLE 2.1

An interpretation of the PPS defined by the three productions

aXb	*bXb*	produce	*cXb*
aYb	*bYc*	produce	*aYc*
YXY		produce	XYX

where $\Sigma = \{a, b, c\}$ and $\Psi = \{X, Y\}$, and the working memory

$$\{aaa, bbb, ccc, abb, bbc, cac\}$$

adds the elements *cbb, abc,* and *aca* to working memory. □

PPSs have proven to be a valuable problem solving tool in many problem domains; two examples are: language specification and knowledge representation. An example of each follows.

EXAMPLE 2.2

Consider a PPS G in which the primitive symbols are divided into a set of non-terminal symbols $N \subseteq \Sigma$, and a set of terminal symbols $\Sigma \sim N$. If all productions are of the form

$$UAV \quad \text{produce} \quad UxV$$

where $\{U, V\}$ are the variables of G, $A \in N$, $x \in \Sigma^*$, and working memory is initially $\{S\}$, where $S \in N$, then G is called a *context-free grammar*. Those working memory elements derived by the productions in G that

contain only terminal symbols comprise the language L(G). In this way, a PPS can specify a language L(G)[75]. □

EXAMPLE 2.3

Consider a PPS with primitive symbols {*man, mortal, is, tom, john*}, variables {X}, and the following production.

X is man produce *X is mortal*

Given the initial working memory

{*john is man, tom is man*}

then the interpretation of this PPS derives the elements *john is mortal*, and *tom is mortal*. The single production ***X is man*** **produce** ***X is mortal*** represents the knowledge that "All men are mortal." In this way, PPSs may represent knowledge, and apply this knowledge to the facts or assertions represented by working memory elements. □

Conventionally, the term *production system* is most commonly associated with knowledge-based expert systems, or psychological modelling applications[76]. Such production systems seldom resemble PPSs[77,78,79,80]. To facilitate the writing of practical production systems, numerous additional features or "extensions" to the PPS formalism are commonly provided by conventional production system architectures. The following six sections describe these features.

2.1.2. Categories and Attributes

In a PPS, working memory elements are arbitrary strings of symbols. In current production systems, however, working memory elements are usually understood to represent objects, where substrings describe *attributes* of those objects. For example, a working memory element

75 Brainerd and Landweber, 1974, pp. 159-161.
76 Davis and King, 1977.
77 ibid.
78 Forgy, 1982.
79 Waterman and Hayes-Roth, 1978.
80 Stefik, *et al.*, 1978.

`expr1 ÷ expr2 expr3`

may logically represent an arithmetic expression with four attributes:

1. an *expression name* with a value of `expr1`,
2. an *arithmetic operator* with a value of ÷,
3. a *left operand* with a value of `expr2`, and
4. a *right operand* with a value of `expr3`.

For clarity, current production systems might represent such a working memory element by

(**name**:expr1 **op**:÷ **left**:expr2 **right**:expr3)

where ***expr1***, ***÷***, ***expr2***, and ***expr3*** are the *attribute values* associated with the attributes ***name***, ***op***, ***left***, and ***right***.

Working memory elements that describe objects of the same type belong to the same *category* and have the same attributes. Because working memory elements belonging to different categories may also have the same attributes, a category name is usually placed at the beginning of each working memory element to avoid confusion[81]. For example, although the working memory elements

(arth_expr **name**:expr1 **op**:÷ **left**:expr2 **right**:expr3)
(bool_expr **name**:expr4 **op**:≠ **left**:expr5 **right**:expr6)

contain the the same set of attributes, they describe different types of objects: arithmetic expressions, and boolean expressions; hence, the categories ***arth_expr*** and ***bool_expr*** at the beginning of each working memory element.

This change to the logical form of working memory elements requires a corresponding change to the form of the patterns that describe them. For example, the patterns

(arth_expr **name**=expr1 **op**=÷ **left**=expr2 **right**=expr3)
(bool_expr **name**=expr4 **op**=≠ **left**=expr5 **right**=expr6)

---

81 A working memory element may be thought of as a Pascal record, where the *category* corresponds to the type of the record, attributes correspond to fields in the record.

consist of a category followed by *predicates* (e.g. *name=expr1*)[82]. Each predicate contains the name of at least one attribute and defines the allowed values for that attribute. For example, the pattern

(arth_expr **name**=expr1 **op**=÷ **left**=**right**)

describes working memory elements from the ***arth_expr*** category that have a ***name*** attribute value equal to ***expr1***, an ***op*** attribute value equal to ÷, and equivalent ***left*** and ***right*** attribute values. Notice that these patterns do not contain variables. The use of variables is discussed below in SECTION 2.1.4. Additionally, not all attributes need to be represented in a pattern. For example, the pattern

(arth_expr **left**=expr2 **right**=expr3)

describes any working memory element from the ***arth_expr*** category with a ***left*** attribute value of ***expr2*** and a ***right*** attribute value of ***expr3***. All other attribute values are ignored. When a pattern contains only a category and no predicates, such as

(arth_expr)

then it describes all working memory elements belonging to that category.

2.1.3. Additional Predicates

The patterns in PPS productions implicitly contain predicates that test strings for equality. For example, consider a PPS working memory element *abc*, where *a*, *b*, and *c* represent strings in Σ^*. An arbitrary PPS pattern *pXq* where *p* and *q* represent strings in Σ^* and *X* is a variable, will describe *abc* if and only if *X* may be instantiated to a string in Σ^* such that

$$(p = a) \wedge (X = b) \wedge (q = c).$$

That is, each PPS pattern implicitly contains an equality predicate for each string or variable. Hence, the use of "=" instead of colons in the patterns of the previous section. Most conventional production system architectures al-

---

82 A *predicate* is a function that returns a value of *true* or *false*.

low patterns to contain other predicates besides "=," such as "≠," "<," ">," "≤," and "≥." For example,

```
(arth_expr  name≥expr1  op=÷  left<0  right≠0)
(bool_expr  name≠expr0  op=≥  substring(left, right))
```

may represent valid patterns, where relational operators (e.g., =, ≠, >, ≥, <, and ≤) are represented by infix notation, and other predicates are represented in prefix notation (e.g., ***substring(left, right)***). Notice that these patterns do not contain variables. The use of variables is discussed in the next section.

2.1.4. Operational Variables and Direct References

PPS productions may contain *operational variables* that are to be consistently instantiated to strings in Σ^*. Such variables are used within a production for any of the following purposes.

1. To compare a substring (i.e., an attribute value) of one working memory element with a substring of another.
2. To compare two different substrings within the same working memory element.
3. To copy a substring from one working memory element into a newly created element.

Most conventional production system architectures also use variables to perform these tasks. FIGURE 2.1(a) contains a typical production using operational variables. However, care must be exercised to insure that every variable in a production is referred to by at least one equality predicate (e.g., ***name*** = ***F***) in that production condition. Even more care must be exercised when a production condition contains negative patterns (cf. SECTION 2.1.5).

To avoid potential problems and confusion, example productions in this thesis use *direct references* in place of variables. For example, FIGURE 2.1(b) contains a production, identical to that of FIGURE 2.1(a), that uses direct references.

Direct references (e.g., ***#1.loc, #1.name,*** and ***#2.name***) refer to the attributes of working memory elements described by the patterns in the pro

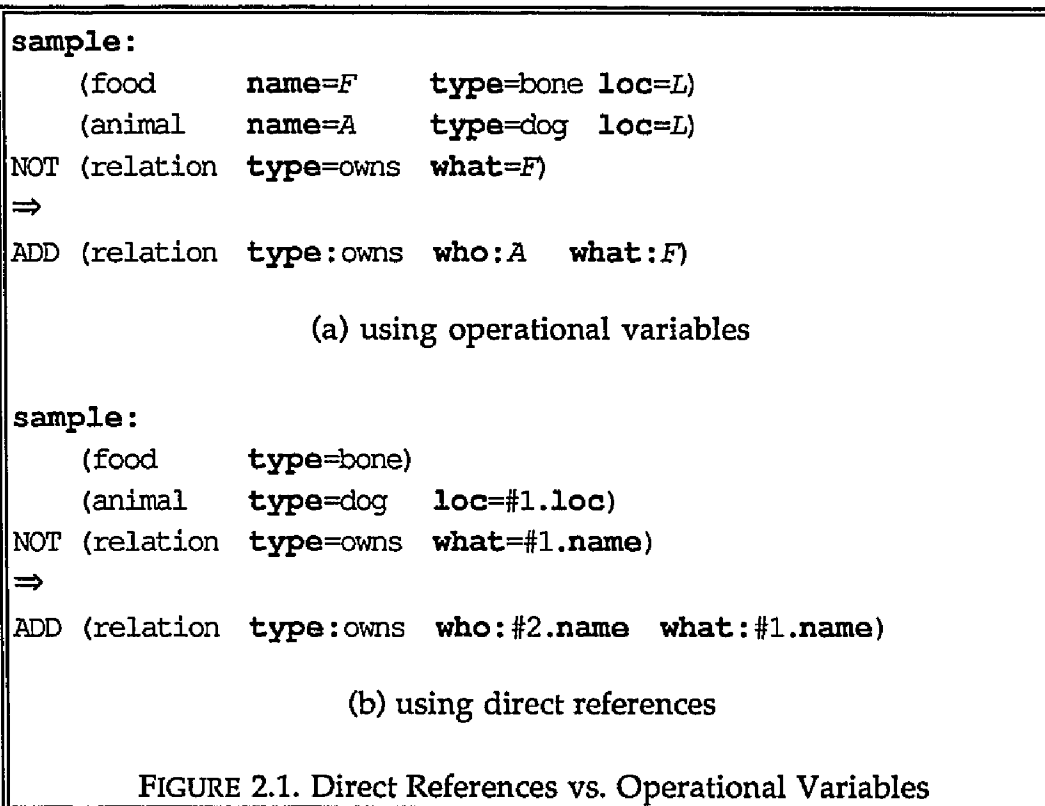


FIGURE 2.1. Direct References vs. Operational Variables

duction. The ordinal part (e.g., ***#1, #2,*** etc.) identifies a working memory element and the attribute part (e.g., ***name, loc,*** etc.) identifies the appropriate attribute of that element. The working memory elements

```
(food name:bone34 type:bone loc:doghouse)
(animal name:pudwuji type:dog loc:doghouse sex:F)
```

are described by the first and second patterns, respectively, because the ***loc*** attribute of the second working memory element is equal to the ***loc*** attribute of the first (i.e., ***loc=#1.loc***). Similarly, the direct reference ***#2.name*** in the action of the production assigns the value ***pudwuji*** from the second working memory element's ***name*** attribute to the ***who*** attribute of the newly created element.

Direct references are used in the thesis in place of operational variables because:

- *Pro forma,* direct references reject dubious comparisons and illegal cross-references between positive and negative patterns (cf. SECTION 2.1.5) that are possible when operational variables are used.
- It is easy to convert direct references into operational variables. It is difficult to formally circumscribe operational variables so that illegal and/or dubious uses are avoided.
- The use of operational variables is a very indirect method of referring to particular attribute values. This suggests that the use of direct references may improve the clarity of productions.

2.1.5. Negated Patterns

In a PPS production every pattern represents a test for the existence of a particular kind of element in working memory. However, in practice it is often necessary to test for the non-existence of such a working memory element. Hence, most conventional production system architectures allow patterns to be *negated* in some way. In this thesis, patterns are negated by preceding them with the word *NOT*. Negated patterns are called *negative patterns*. All other patterns are called *positive patterns*. Negative patterns must *not* describe any elements of working memory if the production which contains them is to be fired. For example, the production in FIGURE 2.1(b) can be fired only if working memory does not contain an element described by the pattern[83].

```
(relation type=owns what=#1.name)
```

Since a negative pattern describes working memory elements that must not exist in working memory, then direct references to negated patterns make no sense. Thus, direct references may only reference positive patterns. This is not to say that negative patterns cannot contain direct references, only that no pattern may reference a negative pattern.

---

83 Technically, a pattern containing direct references makes no sense outside the context of a production condition since the patterns being referenced are undefined. However, for many examples the surrounding context is irrelevant to the point of the example, and the pattern is assumed to exist within the context of a production condition.

2.1.6. Additional Operations

The action of a PPS production always contains exactly one *operation: add a new element to working memory*. In conventional production system architectures, a production action may contain many operations. These operations are not restricted to adding new elements to working memory. Typically, the modification and removal of working memory elements, and various operations which do not affect working memory, such as reading and writing to an I/O device, are allowed.

In this thesis attention will be restricted to the following two types of operations:

- *ADD:* add a new element to working memory. The category and attribute values to assign to the new element are explicitly defined by the parameters of the operation. For example,
  `ADD (arth_expr name:expr1 op:+ left:9 right:#1.left)`
  denotes a valid *ADD* operation.
- *REM:* remove an element from working memory. A single parameter consisting of an ordinal reference to a positive pattern identifies the working memory element to be removed. For example,
  `REM (#1)`
  denotes a valid *REM* operation.

A third operation is also found in some examples:

- *CHG:* change one or more attribute values of a working memory element that matches a positive pattern. The first parameter is an ordinal reference that references a positive pattern, identifying the element to be changed. The remaining parameters list the attributes to be changed along with their new values. For example,
  `CHG (#1, left:#2.left+1, right:#2.right+1)`
  denotes a valid *CHG* operation.

Since the *CHG* operation is implemented by combining an *ADD* and a *REM* operation, it will not be discussed further.

```
feed-a-dog:
    (food   type=meat)
    (animal species=dog  loc=#1.loc  hungry=true)
NOT (animal size>#2.size loc=#1.loc  hungry=true)
⇒
ADD (object type:bone    loc:#1.loc)
REM (#1)
CHG (#2,    hungry:false)
```

FIGURE 2.2. A Sample Production

Most conventional production system architectures incorporate the production extensions of Secs. 2.1.2, 2.1.3, 2.1.5, and 2.1.6; although the actual syntax used varies. The example production of FIGURE 2.2 incorporates all of the above production extensions.

2.1.7. Determinism

Post production systems are non-deterministic. That is, *all* search paths through the problem space defined by a PPS are potentially examined to generate the solution[84]. A search path is defined by the sequence of instantiations fired. To interpret a PPS on a serial machine a strategy must be specified (called the *selection strategy*) that determines which one of many instantiations is fired on each cycle of the interpretation. No invocations are irrevocable, and via backtracking all other search paths may also be examined.

However, many conventional production system architectures are not non-deterministic. That is, a single search path is examined, and a single solution is generated, and no backtracking is performed. For a PPS, all search paths generate equivalent solutions; but, for production systems that allow elements to be removed from working memory this may not be so. Fortunately, for many problem domains a single solution is sufficient[85].

---

84 Where a solution is defined as the final state of working memory.
85 cf. SECTION 2.2.2.1.

2.2. Production System Definitions and Theorems

Previously, many terms related to production systems, such as pattern, attribute, and condition have been introduced informally. This section defines production systems and related concepts more formally.

2.2.1. Structure

2.2.1.1. Working Memory

A production system is composed in part of a set of working memory elements called working memory. In practice each working memory element is understood to represent an object, and consists of a collection of attribute-value pairs that describe the features of that object. Working memory elements that describe objects belonging to the same type or category have the same set of features.

DEFINITIONS

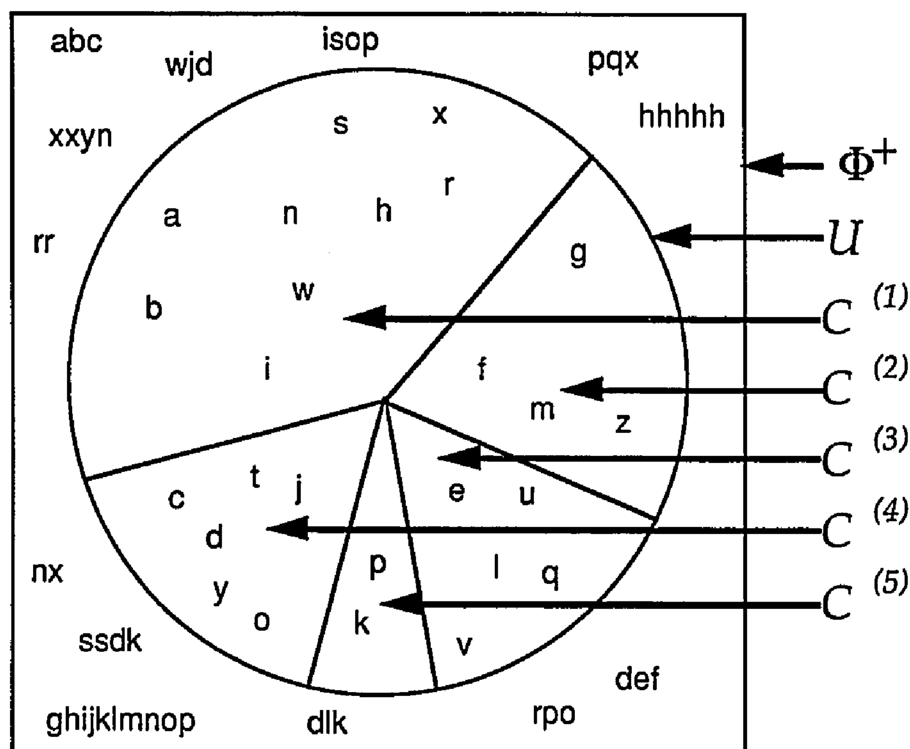
For a given alphabet Φ, $U \subseteq \Phi^+$ is a *working memory element universe,* and elements of U are called *working memory elements.* Each U is partitioned into finitely many disjoint subsets $C^{(1)}, \ldots, C^{(n)}$, called *categories.* Each category $C \in \{C^{(1)}, \ldots, C^{(n)}\}$ is associated with a finite number m of maps $A_1, \ldots, A_m$, called *attributes* that map C into sets $V_1, \ldots, V_m$, of *values;* that is, $A_i : C \to V_i$ for $i \in \{1, \ldots, m\}$, where m, the A_i's, and V_i's all depend on C. A finite set of $W \subseteq U$ is called a working memory. □

In these definitions, the categories, attributes, and values are *primitive* in that they depend upon the particular application.

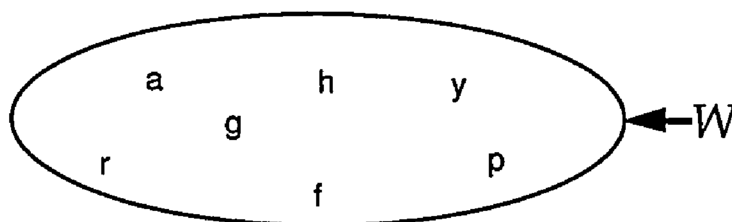
To illustrate the relationship between working memory element universes, categories, and working memories, the following example is presented.

EXAMPLE 2.4

Let $\Phi = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$ be an alphabet, and let $U = \Phi \subseteq \Phi^+$ be a working memory universe. Further, let U be composed of categories $C^{(1)}, \ldots, C^{(5)}$, where $C^{(1)} = \{a,b,h,i,n,r,s,w,x\}$, $C^{(2)} = \{f,g,m,z\}$, $C^{(3)} = \{e,l,q,u,v\}$, $C^{(4)} = \{c,d,j,o,t,y\}$, and $C^{(5)} = \{p,k\}$. The relationships between Φ^+, U, and $C^{(1)}, \ldots, C^{(5)}$ are shown in the following diagram[86]:



A working memory is a finite set $W \subseteq U$. One possible working memory, $W = \{a,f,g,h,p,r,y\}$ is shown in the following diagram:



86 Not all members of the set Φ^+ are displayed.

Notice that W may contain working memory elements from many different categories. □

The actual form of a working memory element (i.e., a string in $U \subseteq \Phi^{+}$) is irrelevant. Any working memory element may be abstracted and treated as a set of attribute values as in the informal description of the previous section. A typical working memory element w is denoted by:

```
(person name:louis age:28 loc:doghouse sex:male)
```

where ***person*** is the category to which w belongs; ***name, age, loc,*** and ***sex*** are the attributes for category ***person;*** and ***louis, 28, doghouse,*** and ***male*** are the values of these attributes. More formally,

$$\begin{aligned}
&\textbf{\textit{person}} \subseteq U \text{ and } w \in \textbf{\textit{person}} \\
&\textbf{\textit{name}} : \textbf{\textit{person}} \rightarrow V_1, \textbf{\textit{name}}(w) = \textbf{\textit{louis}} \\
&\textbf{\textit{age}} : \textbf{\textit{person}} \rightarrow V_2, \textbf{\textit{age}}(w) = \textbf{\textit{28}} \\
&\textbf{\textit{loc}} : \textbf{\textit{person}} \rightarrow V_3, \textbf{\textit{loc}}(w) = \textbf{\textit{doghouse}} \\
&\textbf{\textit{sex}} : \textbf{\textit{person}} \rightarrow V_4, \textbf{\textit{sex}}(w) = \textbf{\textit{male}}
\end{aligned}$$

Note that different attributes (from the same or different categories) may map into the same set of values.

2.2.1.2. Production Memory

In addition to working memory, a production system is also composed of a set of productions called production memory. Each production is composed of a condition and an action, which are in turn composed of patterns and operations, respectively.

2.2.1.2.1. Conditions

DEFINITION

A *k-predicate* p_i^k, $1 \le i \le k$, maps categories $C_1 \times \ldots \times C_k \subseteq U^k$ into $\{true, false\}$ such that for every predicate q mapping categories $C_1 \times \ldots \times C_{i-1} \times C_{i+1} \times \ldots \times C_k$ into $\{true, false\}$ there exist working memory elements $w_{1,k}$, where $w_j \in C_j$ for $j = 1, 2, \ldots, k$, such that $p_i^k(w_1, \ldots, w_k) \neq q\,(w_1, \ldots, w_{i-1},$

$w_{i+1}, \ldots, w_k$). Function p_i^k is of the form $p_i^k(w_1, \ldots, w_k) = f[A_{11}(w_1), \ldots, A_{1m_1}(w_1), \ldots, A_{k1}(w_k), \ldots, A_{km_k}(w_k)]$, where $A_{i1} \ldots A_{im_i}$ and $V_{i1} \ldots V_{im_i}$ are the associated attributes and value sets of C_i, $1 \le i \le k$, and $f: V_{11} \times \ldots \times V_{1m_1} \times \ldots \times V_{k1} \times \ldots \times V_{km_k} \to \{true, false\}$. □

That is, a k-predicate takes working memory elements from specific categories as arguments. For example, in FIGURE 2.2, ***species=dog*** denotes the 3-predicate p_2^3 that maps categories ***food*** × ***animal*** × ***animal*** into {*true*, *false*} such that for all $w_1 \in$ ***food***, $w_2 \in$ ***animal***, and $w_3 \in$ ***animal***, $p_2^3(w_1, w_2, w_3)$ iff ***species***(w_2) = ***dog***.

A conjunction of k-predicates $a_i^k \wedge b_i^k$ defined on the same domain is also a k-predicate defined on that domain. For example, in FIGURE 2.2,

```
(animal species=dog loc=#1.loc hungry=true)
```

denotes a 3-predicate p_2^3 defined on categories ***food*** × ***animal*** × ***animal*** such that for all $w_1 \in$ ***food***, $w_2 \in$ ***animal***, and $w_3 \in$ ***animal***, $p_2^3(w_1, w_2, w_3)$ iff ***species***(w_2) = ***dog*** ∧ ***loc***(w_2) = ***loc***(w_1) ∧ ***hungry***(w_2) = ***true***.

In practice, the "*k*-" is often omitted from the term *k-predicate* when the value of k is either irrelevant or obvious.

DEFINITIONS

A k-predicate p_i^k is said to be *dependent on argument* j, $1 \le j \le k$, if for every q mapping categories $C_1 \times \ldots \times C_{j-1} \times C_{j+1} \times \ldots \times C_k$ into {*true*, *false*} there exist elements $w_{1,k}$, where $w_h \in C_h$ for $h = 1, 2, \ldots, k$, such that $p_i^k(w_1, \ldots, w_k) \ne q(w_1, \ldots, w_{j-1}, w_{j+1}, \ldots, w_k)$. Otherwise, p_i^k is said to be *independent of argument* j. □

That is, a k-predicate is dependent on argument j, if the value of the k-predicate is affected by its jth argument. If the jth argument cannot affect the result, then the predicate is independent of argument j. For example, the 3-predicate in FIGURE 2.2 denoted by

```
(animal species=dog loc=#1.loc hungry=true)
```

is dependent upon arguments 1 and 2, but is independent of argument 3. Notice that a k-predicate p_i^k is, by definition, always dependent upon its ith argument.

DEFINITIONS

A *one-element k-predicate* p_i^k is dependent on argument j iff $j = i$. A *many-element k-predicate* q_i^k is dependent on at least one argument $j \neq i$, $1 \leq j \leq k$. □

One-element k-predicates test the attribute values of individual working memory elements (e.g., ***species = dog***); they contain no direct references. Many-element k-predicates compare attribute values of multiple working memory elements (e.g., ***loc = #1.loc***); they must contain direct references.

DEFINITIONS

Let p_i^k be a k-predicate that is a conjunction of smaller k-predicates. The conjunction of all one-element k-predicates in p_i^k is called the *one-element component of* p_i^k and is denoted by $p_i^{k'}$. Similarly, the conjunction of all many-element k-predicates in p_i^k is called the *many-element component of* p_i^k and is denoted by $p_i^{k''}$. □

For example, if

```
(animal species=dog loc=#1.loc  hungry=true)
```

denotes p_i^k, the 3-predicate from FIGURE 2.3, then

```
(animal species=dog hungry=true)
```

denotes $p_i^{k'}$, the one-element component of p_i^k, and

```
(animal loc=#1.loc)
```

denotes $p_i^{k''}$, the many-element component of p_i^k.

Notice that any one-element k-predicate p_i^k has a trivial many-element component; that is, $p_i^k = p_i^{k'}$ and $p_i^{k''}$ is a tautology. Similarly, any many-element k-predicate p_i^k has a trivial one-element component; that is, $p_i^k = p_i^{k''}$ and $p_i^{k'}$ is a tautology.

LEMMA 2.1

A k-predicate $p_i^k(w_{1,k}) = p_i^{k}{}'(w_{1,k}) \wedge p_i^{k}{}''(w_{1,k})$.

Proof. By definition, a k-predicate is a conjunction of one or more k-predicates defined on the same domain; $p_i^k(w_{1,k}) = p_{i1}^k(w_{1,k}) \wedge \ldots \wedge p_{in}^k(w_{1,k})$. The "$\wedge$" operator is commutative, and the predicates on the RHS of this expression can appear in any order. Therefore, one can assume that all m one-element predicates are grouped to the left (i.e., $p_{i1}^k, \ldots, p_{im}^k$), and that all $n - m$ many-element predicates are grouped to the right (i.e., $p_{im+1}^k, \ldots, p_{in}^k$). This implies that the one-element component $p_i^{k}{}'(w_{1,k}) = p_{i1}^k(w_{1,k}) \wedge \ldots \wedge p_{im}^k(w_{1,k})$ and the many-element component $p_i^{k}{}''(w_{1,k}) = p_{im+1}^k(w_{1,k}) \wedge \ldots \wedge p_{in}^k(w_{1,k})$. Substituting into the original equation, $p_i^k(w_{1,k}) = p_i^{k}{}'(w_{1,k}) \wedge p_i^{k}{}''(w_{1,k})$. □

DEFINITION

A k-predicate p_i^k such that both $p_i^{k}{}'$ and $p_i^{k}{}''$ are tautologies is called a *vacuous k-predicate.* □

In practice, all vacuous k-predicates are ignored.

LEMMA 2.2

For every non-vacuous one-element k-predicate p_i^k defined on some $C_1 \times \ldots \times C_k \subseteq U^k$ there exists a one-element 1-predicate q_1^1 such that $p_i^k(w_{1,k}) = q_1^1(w_i)$ for all $w_{1,k} \in C_1 \times \ldots \times C_k$.

Proof. By the definition of a non-vacuous one-element k-predicate, there must exist such a q_1^1, the projection of p_i^k onto C_i. □

For any k-predicate p_i^k, the one-element 1-predicate corresponding to $p_i^{k}{}'$, as defined in LEMMA 2.1, is denoted by $p_i^k\!\downarrow$. For example, let p_2^3 be the 3-predicate in FIGURE 2.2, denoted by

(food type=meat)

such that for all $w_1 \in$ ***food***, $w_2 \in$ ***animal***, and $w_3 \in$ ***animal***, $p_2^3(w_1, w_2, w_3)$ iff ***food***(w_1) = ***meat***. Then, for all $w_1 \in$ ***food***, $p_2^3\!\downarrow(w_1)$ iff ***food***(w_1) = ***meat***.

DEFINITIONS

A *k-pattern* is a pair $\langle s, p_i^k \rangle$, where $s \in \{+, -\}$ is called the *sign*, and p_i^k is a k-predicate defined on some categories $C_1 \times \ldots \times C_k$. A k-pattern with a sign of + is called *positive k-pattern*. Similarly, a k-pattern with a sign of – is called a *negative k-pattern*. A k-pattern is said to be *satisfied* by a working memory W if there exists $w_{1,k} \in (C_1 \cap W) \times \ldots \times (C_k \cap W)$ such that $p_i^k(w_{1,k}) = true$. □

In practice, the "*k*-" is often omitted from the term *k-pattern* when the value of k is either irrelevant or obvious. In examples, negative patterns are denoted by k-predicates preceded by the word *NOT*, such as

```
NOT (animal  size>#2.size  loc=#1.loc  hungry=true)
```

while positive patterns, such as

```
    (animal  species=dog  loc=#1.loc  hungry=true)
```

are not preceded by the word *NOT*.

DEFINITIONS

A set of k-patterns, $c = \{\langle s_1, p_1^k \rangle, \langle s_2, p_2^k \rangle, \ldots, \langle s_k, p_k^k \rangle\}$ is a *condition* if, for some $C_1 \times \ldots \times C_k \in U^k$, and for all $i = 1, \ldots, k$:

1. p_i^k is defined on $C_1 \times \ldots \times C_k$, and
2. p_i^k is dependent upon argument j implies that $j = i$ or $s_j = +$.

Condition c is *satisfied* by working memory W if $\forall \langle +, p_i^k \rangle \in c, \exists w_{1,k} \in W^k$ such that $p_i^k(w_{1,k}) = true$, and $\forall \langle -, p_i^k \rangle \in c, \forall w_i \in C_i, p_i^k(w_{1,k}) = false$. □

In other words, a condition is a sequence of k-patterns such that all k-predicates are defined on the same categories, and all direct references refer to positive patterns in the condition.

For example, the patterns

```
    (food    type=meat)
    (animal  species=dog   loc=#1.loc  hungry=true)
NOT (animal  size>#2.size  loc=#1.loc  hungry=true)
```

denote a condition; but, the patterns

```
    (food    type=meat)
    (animal  species=dog   loc=#3.loc   hungry=true)
NOT (animal  size>#6.size  loc=#1.loc   hungry=true)
```

do not denote a condition because negative patterns and non-existent patterns are referenced (e.g., ***#3.loc*** and ***#6.size*** respectively).

DEFINITION

A *satisfying list* for a condition is a list of working memory elements $w_{1,k}$ for which the k-predicates in the positive patterns of that condition evaluate to *true*. The ith working memory element in a satisfying list is said to *match* the ith pattern in the corresponding condition. □

For example,

```
⟨ (food   type:meat    loc:doghouse),
  (animal name:pudwuji species:dog loc:doghouse hungry:true),
  – ⟩
```

is a satisfying list[87] for the following condition from FIGURE 2.2:

```
    (food    type=meat)
    (animal  species=dog   loc=#1.loc   hungry=true)
NOT (animal  size>#2.size  loc=#1.loc   hungry=true)
```

2.2.1.2.2. Actions

In addition to a condition, every production contains an action that modifies the contents of working memory. Each action is composed of one or more operations that sequentially perform small modifications to working memory, such as adding or removing an element.

DEFINITION

An *operation* is a function $o : C_1 \times \ldots \times C_k \times 2^{U} \rightarrow 2^{U}$. □

---

87 The "–" symbol instantiation denotes that no working memory element matches the third pattern.

That is, given a satisfying list $w_{1,k}$ and a working memory W, an operation o uses information in $w_{1,k}$ to modify W, and returns the resulting working memory. This thesis is concerned with only two types of operations: *add* and *remove*.

DEFINITIONS

An operation o is an *add* operation if $o(w_{1,k}, W) = W \cup \{w\}$ for some w in U. Similarly, an operation o is a *remove* operation if $o(w_{1,k}, W \cup \{w\}) = W$ for some $w \in \{w_1, \ldots, w_k\} \cap W$. □

As the name implies, an add operation adds a new element to working memory. For example, an add operation from FIGURE 2.2

```
ADD (food    type:bone    loc:#1.loc)
```

instantiates ***#1.loc*** to ***doghouse***, based upon the satisfying list above, and adds the working memory element denoted by

```
(object  type:bone  loc:doghouse)
```

to working memory. Similarly, a remove operation removes an existing element from working memory. The remove operation from FIGURE 2.2

```
REM (#1)
```

would remove from working memory the first element in the satisfying list (i.e., the element that matches the first pattern in the condition):

```
(food    type:meat  loc:doghouse)
```

DEFINITIONS

An *action* is a function $a : C_1 \times \ldots \times C_k \times 2^U \rightarrow 2^U$ of the form $a(w_{1,k}, W) = o_m(w_{1,k}, o_{m-1}(w_{1,k}, \ldots o_2(w_{1,k}, o_1(w_{1,k}, W)) \ldots))$ where $o_1, \ldots, o_m$ are operations that map $C_1 \times \ldots \times C_k \times 2^U \rightarrow 2^U$. □

That is, an action performs a composition of operations on a satisfying list of working memory elements and a working memory to produce a new working memory.

DEFINITIONS

A *production on* U is a triple $\langle n, c, a\rangle$ where n is called the *production name*, c is a condition defined on domain $C_1 \times \ldots \times C_k$, and a is an action defined on domain $C_1 \times \ldots \times C_k \times 2^U$. The set of all productions on U is denoted by $\mathbb{P}(U)$. A finite set of productions $\mathcal{P} \subseteq \mathbb{P}(U)$ is called a *production memory on* U if $\forall \langle n_1, c_1, a_1\rangle, \langle n_2, c_2, a_2\rangle \in \mathcal{P}, n_1 \neq n_2$. □

In practice, n is often used to refer to a production $\langle n, c, a\rangle$. A production $\langle$*name, condition, action*$\rangle$ is denoted by *name : condition* $\Rightarrow$ *action*. A sample production may be found in FIGURE 2.2.

DEFINITION

A *production system* $\langle \mathcal{P}, W\rangle$ consists of a production memory $\mathcal{P} \subseteq \mathbb{P}(U)$ for some U and a working memory $W \subseteq U$. □

In conventional programming terms a production memory $\mathcal{P}$ may be thought of as a program. Similarly, a working memory W may be thought of as the data input to $\mathcal{P}$.

DEFINITIONS

An *instantiation on* $\langle \mathcal{P}, W\rangle$ is a pair $\langle\langle n, c, a\rangle, w_{1,k}\rangle$, where $\langle n, c, a\rangle \in \mathcal{P}$, and $w_{1,k} \in W^k$ is a satisfying list for c. An instantiation is *fired* by replacing W with $a(w_{1,k}, W)$. A set of instantiations on $\langle \mathcal{P}, W\rangle$ is called a *conflict set* of $\langle \mathcal{P}, W\rangle$. □

An instantiation describes one way that the patterns in a production condition can be matched with elements of working memory, and contains all the information necessary for the action of that production to modify working memory. Instantiations are also called *instantiated productions*. FIGURE 1.3 denotes a sample instantiation.

Firing an instantiation applies the action of the satisfied production to working memory. When the term *fire* references a production, it is intended that an instantiation of that production is actually being fired.

2.2.2. Interpretation

Informally, a production system interpreter finds those productions that have conditions satisfied by working memory, and applies their actions to working memory. However, the actual *interpretation scheme* used varies from one production system architecture to another. The interpretation scheme, sometimes called the *search strategy,* consists of the *backtracking strategy, evaluation strategy,* and *selection strategy* used by the interpreter. Each of these concepts is discussed below.

2.2.2.1. Backtracking Strategy

Post production systems are non-deterministic and may search all paths of a problem space to find a solution. Backtracking is used to implement non-deterministic production systems on serial machines. By saving the state of working memory before following a particular search path (i.e., before firing an instantiation), one may later backtrack to that state and follow a different search path (i.e., fire a different instantiation). In this way, all search paths may be examined. Because the time and space costs of backtracking are non-trivial, most conventional production system architectures are deterministic and do not backtrack. This is the case for the palimpsest production system architecture described herein.

2.2.2.2. Evaluation Strategy

Another piece of the interpretation scheme for a production system architecture is the direction of evaluation, or *evaluation strategy*. There are two primary evaluation strategies: *forward-chaining* and *backward-chaining*. Additionally, there are numerous variations on these, such as *bi-directional* and *means-ends analysis* techniques. Each is discussed below.

Forward-chaining or *data-driven* evaluation searches production systems for productions with satisfied conditions and applies their actions to working memory, one at a time, until no new conditions can be satisfied. However, time may be wasted satisfying the conditions and applying the actions of productions that do not contribute to the desired problem solution. A forward-

chaining evaluation strategy is desirable when the effect of production actions on other production conditions is poorly defined, or when the number of problem solutions is large.

Backward-chaining or *goal-driven* evaluation also searches production systems for satisfied production conditions, but that search is directed to satisfy the condition of a goal production. If working memory does not satisfy the condition of the current goal production, and the action of another production helps to satisfy the condition of the goal production, then the interpreter designates that new production as the goal production. However, time may be wasted satisfying goals that do not contribute to the eventual problem solution. A backward-chaining evaluation strategy is desirable when the effect of production actions on other production conditions is known, and when the number of problem solutions (i.e., goals) is small.

Bi-directional evaluation attempts to combine the simplicity of forward-chaining evaluation with the directed search capability of backward-chaining evaluation. Both forward-chaining and backward-chaining evaluation are applied simultaneously, in the hopes that the evaluations meet somewhere in the middle of the problem space.

Means-ends analysis evaluation is a heuristic technique that can be used to search a problem space more efficiently. When the difference between the current state of the interpreter and a goal state can be quantified, the decision to apply the action of a production with a satisfied condition depends upon that action's ability to reduce that difference.

The production system architecture defined in CHAPTER 4 is essentially forward-chaining. However, this architecture can use *any* of the above evaluation strategies, singly or in combination (cf. SECTION 6.1).

2.2.2.3. Selection Strategy

Within the constraints imposed by a given evaluation strategy, a *selection strategy* determines the exact search path followed by a production system interpreter for a production system. Because problem solutions may depend

upon the search path followed, and because identical problem solutions may be found by following search paths of different lengths, the choice of an appropriate selection strategy is crucial to the correctness and efficiency of a production system interpreter.

DEFINITION

A *match function* M maps $2^{\mathbb{P}(U)} \times 2^{U} \to 2^{\mathbb{C}(U)}$, where $\mathbb{C}(U)$ is the set of all instantiations on $\langle \mathbb{P}(U), U \rangle$, such that $M(\mathcal{P}, W)$ is a conflict set of $\langle \mathcal{P}, W \rangle$, and $|M(\mathcal{P}, W)| > 0$ iff an instantiation on $\langle \mathcal{P}, W \rangle$ exists. □

A match function takes a production system as its argument, finds instantiations on that production system and returns them in a conflict set.

DEFINITION

A *conflict resolution function* R maps $2^{\mathbb{C}(U)} \to \mathbb{C}(U)$ such that $R(S) \in S$ for $S \subseteq \mathbb{C}(U)$. □

A conflict resolution function takes a conflict set as its argument and returns one instantiation from that conflict set. For example, given some conflict set, a conflict resolution function might return an instantiation containing the production with the most complex condition.

DEFINITION

A pair $\langle M, R \rangle$, where M is a match function and R is a conflict resolution function, defines a *selection strategy*. □

A selection strategy defines the process by which a deterministic, forward chaining, production system interpreter, defined by ALGORITHM 2.1, selects a production to be fired on each cycle.

ALGORITHM 2.1

Deterministic production system interpretation using selection strategy $\langle M, R \rangle$.

Input. A production system $\langle \mathcal{P}, W \rangle$.

Output. A modified working memory.

Method. The state of the algorithm is represented by W. Perform steps (1) to (3).

1. *Match Phase*: Determine conflict set $S = M(\mathcal{P}, W)$ and **go to** step (2).
2. *Conflict Resolution Phase*: If S is empty then output W and **halt**. Otherwise, select an instantiation $I = R(S)$ and **go to** step (3).
3. *Act Phase*: Fire instantiation $I = (\langle n, c, a\rangle, w_{1,k})$ by assigning $a(w_{1,k}, W)$ to W, and **go to** step (1). □

This algorithm repeatedly finds, selects, and fires instantiations.

DEFINITION

ALGORITHM 2.1 is called the *recognize-act cycle*. □

2.2.3. Procedural Control

Procedural control information for production systems specifies the order in which instantiated productions can be fired. The most common methodology for specifying and implementing procedural control is to write a production system to take advantage of the conflict resolution function[88,89]. Unfortunately, as described in SECTION 1.2.2, such control strategies destroy the modularity of production systems. Another approach augments the production system with separate control information, in the form of a control language, that restricts the productions that can be fired on any given cycle. Such "controlled production systems" are addressed in this section.

2.2.3.1. Controlled Production Systems

A definitive description of controlled production systems and their inherent advantages over other control strategies is found elsewhere[90]. This section presents a brief introduction to controlled production systems that conforms to the notation of this thesis.

88 Rychener, 1979.
89 McDermott and Forgy, 1977.
90 Georgeff, 1982.

DEFINITIONS

Let $N(\mathcal{P})$ be the set of all production names found in a production memory $\mathcal{P}$. Then, $K \subseteq N(\mathcal{P})^*$ is called a *control language* of $\mathcal{P}$. A context-free grammar G_K, where $K = L(G_K)$, is called a *control grammar* of $\mathcal{P}$. □

A control language is defined on an alphabet containing production names. Strings in a control language specify the "legal" sequences of production firings for a production system. Herein, all control languages are specified by context-free control grammars.

DEFINITIONS

Given a control language $K \subseteq N(\mathcal{P})^*$ and a string $s \in N(\mathcal{P})^*$ that denotes a sequence of production firings, production $\langle n, c, a \rangle \in \mathcal{P}$ is a *legal production* if $sn\alpha \in K$ for some $\alpha \in N(\mathcal{P})^*$. An instantiation is a *legal instantiation* if it contains a legal production. □

For example, the following control grammar[91]

$$S \rightarrow p_1 \, S \, p_2$$
$$S \rightarrow \varepsilon$$

specifies that all sequential firings of production p_1 must be followed by exactly the same number of firings of production p_2. Initially, only production p_1 is legal. After firing p_1, then both p_1 and p_2 are legal. After firing the sequence of productions denoted by the string $p_1 \, p_1 \, p_1 \, p_2$, only production p_2 is legal.

DEFINITION

A *controlled production system* $(\langle \mathcal{P}, W \rangle, G_K)$ consists of a production system $\langle \mathcal{P}, W \rangle$ and a control grammar G_K of $\mathcal{P}$. □

A controlled production system uses a control language to impose procedural control allowing the production system interpreter to fire only those instantiations containing legal productions.

---

91 Herein, all non-terminal grammar symbols are denoted by upper-case letters, and all terminal grammar symbols are denoted by lower-case letters.

EXAMPLE 2.5

Consider a production memory containing two productions that install nuts onto threaded bolts:

```
load_nut:
     (bolt   filled=false)
     (nut    location=unused)
⇒
CHG  (#2,    location:#1.location)

turn_nut:
     (bolt   filled=false)
     (nut    location=#1.location)
⇒
CHG  (#1,    filled:true)
```

Production ***load_nut*** places an unused nut onto an empty bolt, and ***turn_nut*** screws the nut onto the bolt. It is assumed that there are at least as many nuts as there are threaded bolts. Clearly, every call to ***turn_nut*** must be preceeded by a call to ***load_nut***. If the goal of this production memory is to install nuts onto all empty bolts, then this iterative sequencing of production firings can be specified by the following control grammar

INSTALL_NUTS → load_nut turn_nut INSTALL_NUTS
INSTALL_NUTS → stop_installing_nuts

where ***stop_installing_nuts*** is a production with an empty action

```
stop_installing_nuts:
NOT  (bolt   filled=false)
⇒
```

that is satisfied only when nuts have been installed on all bolts. Because the first symbol of any string in this control language must be either ***load_nut*** or ***stop_installing_nuts***, then the first production fired must be one of these two productions. Given an initial working memory of

```
(bolt location:loc27  filled:false)
(nut  location:unused)
```

only ***load_nut*** has a satisfied condition. After firing ***load_nut***, the only legal production, according to the control grammar, is ***turn_nut***. After firing ***turn_nut*** both ***load_nut*** or ***stop_installing_nuts*** are once again le-

gal; however, only ***stop_installing_nuts*** has a satisfied condition. After ***stop_installing_nuts*** fires, no productions are legal, and interpretation should halt. □

A larger example of a controlled production system is presented in Appendix B. ALGORITHM 2.2 defines the interpretation of controlled production systems.

ALGORITHM 2.2

Deterministic controlled production system interpretation using selection strategy $\langle M, R\rangle$.

Input. A controlled production system $(\langle \mathcal{P}, W\rangle, G_K)$.

Output. A modified working memory.

Method. The state of the algorithm is represented by the pair $\langle s, W\rangle$, where s is an initially empty string in $N(\mathcal{P})^*$. Perform steps (1) to (4).

1. *Control Phase*: Determine the set of *legal productions* $P = \{\langle n, c, a\rangle \mid \langle n, c, a\rangle \in \mathcal{P}$, and $sn\alpha \in L(G_K)$ for some string $\alpha \in N(\mathcal{P})^*\}$ and **go to** step (2).
2. *Match Phase*: Determine conflict set $S = M(P, W)$ and **go to** step (3).
3. *Conflict Resolution Phase*: If S is empty then output W and **halt.** Otherwise, select an instantiation $I = R(S)$ and **go to** step (4).
4. *Act Phase*: Fire instantiation $I = (\langle n, c, a\rangle, w_{1,m})$ by assigning $a(w_{1,m}, W)$ to W, assign sn to s, and **go to** step (1). □

Interpretation of controlled production systems is identical to the interpretation of conventional production systems, except that the match phase only searches for instantiations of "legal" productions. Notice that ALGORITHM 2.2 reduces to ALGORITHM 2.1 when $K = N(\mathcal{P})^*$; that is, when the sequence of production firings is unconstrained, and no procedural control is specified by the control language.

DEFINITION

A controlled production system $\langle\langle\mathcal{P}, W\rangle, G_K\rangle$ is *control free* if $L(G_K) = N(\mathcal{P})^*$ and the working memory computed by ALGORITHM 2.2 is independent of selection strategy $\langle M, R\rangle$. □

That is, a controlled production system is control free if the control language specifies no procedural control, and productions do not take advantage of the conflict resolution strategy to impose procedural control. Because $K = N(\mathcal{P})^*$ in a control free controlled production system, a conventional, "uncontrolled" production system can also be said to be control free if it does not take advantage of the conflict resolution strategy to impose procedural control.

2.2.3.2. Examples of Common Control Constructs

This section demonstrates how many of the more important control constructs can be specified in a controlled production system[92]. Of course, all of the constructs described may be combined to produce more complex control constructs. For the purposes of discussion, assume that the production memory in question contains productions with names $p_1, p_2, \ldots, p_n$.

2.2.3.2.1. Direct Sequencing

Direct sequencing control specifies a sequence of productions to be fired. For example, one may need to specify that production p_9 must fire, followed immediately by p_3 and p_6. Such a sequence is specified by a placing the string $p_9\, p_3\, p_6$ in the control grammar.

$$\text{SEQUENCE} \rightarrow p_9\, p_3\, p_6$$

where p_9, p_3, and p_6 are terminal symbols, and *SEQUENCE* is a non-terminal symbol.

92 A complete description of these control constructs and their importance in AI programming is presented in Rychener, 1979.

2.2.3.2.2. Fall-Back Control and Held Result Usage

Fall-back control resembles a subroutine call and return. Productions associated with a particular process pass control to a different group of productions associated with some subprocess. When that subprocess has been completed, control returns to the original process, which does not directly use the results generated by the subprocess. Held result usage is just fall-back control in which the original process *does* directly use the results generated by the subprocess, but allows that subprocess to terminate before processing those results. Fall-back control resembles a procedure call in Pascal or Ada, and held result usage resembles a function call. For example, in the control grammar

$$\text{MAINPROCESS} \rightarrow p_1\, p_2\, \text{SUBPROCESS}\, p_3\, p_4$$
$$\text{SUBPROCESS} \rightarrow p_5\, p_6$$

MAINPROCESS specifies that p_1 and p_2 should be fired, in sequence, and followed by the processing of *SUBPROCESS*. In this case, *SUBPROCESS* merely involves firing p_5 and p_6 in sequence. After *SUBPROCESS* is completed (i.e., p_6 is fired) control then returns to *MAINPROCESS* at the point immediately following *SUBPROCESS*. Productions p_3 and p_4 are then fired, in sequence, to complete *MAINPROCESS*.

2.2.3.2.3. Selection

Selection allows any one of a number of productions to fire. For example, the following control grammar

$$\text{SELECT} \rightarrow p_1$$
$$\text{SELECT} \rightarrow p_2$$
$$\text{SELECT} \rightarrow p_3$$

specifies that either p_1, p_2, or p_3, should fire. Selection is used to choose a production based solely upon the satisfiability of its condition; no procedural constraints are imposed.

2.2.3.2.4. Iteration

Iterations of single productions, sequences or even complex control constructs can be specified using recursive grammar rules, such as *INSTALL_NUTS* in EXAMPLE 2.5, and the following:

$$\text{LOOP} \rightarrow p_1 \text{ COMPLEX_CONSTRUCT } p_2 \text{ LOOP}$$
$$\text{LOOP} \rightarrow \text{stop_loop}$$

where ***stop_loop*** is a production with an empty action that specifies the termination condition for the loop. Initially, both p_1 and ***stop_loop*** are allowed to fire. If the condition of p_1 is satisfied, then the loop defined by the first rule is entered; otherwise, the condition of ***stop_loop*** should be satisfied and the loop terminates. If p_1 fires, then a sequence of productions defined by ***COMPLEX_CONSTRUCT*** will fire, then p_2 will fire. At this point, both p_1 and ***stop_loop*** are again allowed to fire, and the loop either terminates or begins another iteration.

2.2.3.2.5. Modules and Hierarchies

A complex control construct (i.e., a portion of a control grammar that could stand alone) and the productions it references can be thought of as a production module that performs a single, well-defined task. The name of this module is the goal symbol of its control grammar. Such modules can be called from within other modules, via fall-back control or held result usage. Using this scheme, complex production module hierarchies can be constructed that resemble the structure of procedural programs such as Pascal and Ada[93], and facilitate top-down design methodologies. The control construct *INSTALL_NUTS* from EXAMPLE 2.5 is a production module that can be called by the following module

CHANGE_FLAT → remove_nuts SWAP_TIRES INSTALL_NUTS
CHANGE_FLAT → call_towtruck FIX_TIRE PAY_MECHANIC

[93] In fact, a production system language can be defined that enforces module nesting and scoping identical to that found in the procedural languages.

where ***remove_nuts,*** and ***call_towtruck*** are productions and *SWAP_TIRES, INSTALL_NUTS, FIX_TIRE,* and *PAY_MECHANIC* are lower level production modules that perform the actions suggested by their names.

Productions define a single action that can be applied whenever a single condition is satisfied. Production modules, on the other hand, generally represent larger, dynamic plans of action, where the exact plan followed is determined dynamically, based upon the satisfiability of various conditions. In the example above there are two plans for changing a flat tire. The first plan is applied if the condition of ***remove_nuts*** is satisfied; and the second plan is applied if the condition of ***call_towtruck*** is satisfied. In general, the condition of the leftmost production (or module) in any plan acts as a condition for that entire plan. When the condition of the first production in a plan is insufficient to determine whether or not that plan can be applied, a new production should be constructed that can make that determination and inserted at the beginning of the plan. A disjunction of the conditions for all the plans in a module acts as the condition for that module.

2.2.3.2.6. Concurrent Control Constructs

The efficient implementation of various other control constructs requires a production system architecture that can interpret multiple productions or production modules concurrently. Like most production system architectures, palimpsest parsers currently allow the interpretation of only one production or production module at any time. However, concurrent execution of palimpsest parsers may be implemented in the near future. For this reason, some concurrent control constructs found in the literature are described briefly below[94].

- *Direct Result Usage:* similar to held result usage, except the calling process makes use of the results of the called process as they are generated.

94 Rychener, 1977.

- *Fork-Join:* A problem is solved by decomposing it into subproblems, and assigning the solution of the subproblems to a number of concurrent processes. When all subproblems have been solved, the various results are collected and assembled by the original process.
- *Parallel Iteration:* The productions in the body of a loop iterate concurrently.

Of course, if absolutely necessary, these three control constructs can be implemented serially, but the size of that required control grammar grows exponentially with the number of productions or modules that must be interpreted concurrently.

2.3. Scope of Controlled Production Systems

It has been shown that Post production systems are computationally equivalent to deterministic Turing machines. This section demonstrates that controlled production systems, as defined above, are also computationally equivalent to deterministic Turing machines.

THEOREM 2.2

For every deterministic Turing machine there exists an equivalent controlled production system.

Proof. By construction. Let $M = (S, \Sigma, \delta, s, \$, Y)$ be a deterministic Turing machine, where S is a finite set of states, Σ is the tape alphabet, $s \in S$ is the start state, $\$ \in \Sigma$ is the blank symbol, $Y \subseteq S$ is the set of accepting states, and δ is any partial function from $S \times \Sigma \to S \times \Sigma \times \{left, right, stay\}$[95]. Any such M may be represented by a working memory W as follows:

For convenience, let tape positions be assigned consecutive integer addresses, where position 0 is the initial position of the read/write head. The infinite tape may be represented by a set T of pairs of the form ⟨*<address>*, *<symbol>*⟩, where *<address>* is a tape address, *<symbol>* $\in \Sigma$, ⟨*<address>*, *<symbol>*⟩ $\notin T \Rightarrow$ *<symbol>* = \$, and (⟨*<address1>*, *<symbol>*⟩ $\in T \wedge$

95 Savitch, 1982, p. 77.

⟨*<address2>*, *<symbol>*⟩ ∈ *T*) ⇒ *<address1>* ≠ *<address2>*. Denote members of *T* by working memory elements of the form

```
(tape addr:<address> symbol:<symbol>)
```

A configuration of the Turing machine may be represented by a pair *c* of the form ⟨*<state>*, *<address>*⟩, where *<state>* ∈ *S*, *<address>* is a tape address. Denote a configuration by a working memory element of the form

```
(config state:<state> head:<address>)
```

Initially, this element is ⟨*s*, *0*⟩, denoted by

```
(config state:s head:0)
```

Represent δ by a set *I* as follows. For all *<state1>*, *<state2>* ∈ *S* and *<symbol1>*, *<symbol2>* ∈ Σ such that δ(*<state1>*, *<symbol1>*) = (*<state2>*, *<symbol2>*, *<dir>*), where *<dir>* ∈ {*left*, *right*, *stay*}, *I* contains 5-tuples of the form ⟨*<state1>*, *<symbol1>*, *<state2>*, *<symbol2>*, *<dir>*⟩. Denote the members of *I* by working memory elements of the form

```
(instr on_state:<state1> to_state:<state2> read:<symbol1>
       write:<symbol2> move:<dir>)
```

A deterministic Turing machine interpreter may be implemented by the following procedure:

loop
    if *c* = ⟨*<state>*, *<address>*⟩
        and not ⟨*<address>*, *<symbol>*⟩ ∈ *T* **then**
            T := {⟨*<address>*, $⟩} ∪ *T*;
    else if *c* = ⟨*<state>*, *<address>*⟩
        and ⟨*<address>*, *<symbol>*⟩ ∈ *T*
        and ⟨*<state>*, *<symbol>*, *<state2>*, *<symbol2>*, *left*⟩ ∈ *I* **then**
            T := (*T* ~ {⟨*<address>*, *<symbol>*⟩}) ∪ {⟨*<address>*, *<symbol2>*⟩};
            c := ⟨*<state2>*, *<address>* − 1⟩;
    else if *c* = ⟨*<state>*, *<address>*⟩
        and ⟨*<address>*, *<symbol>*⟩ ∈ *T*
        and ⟨*<state>*, *<symbol>*, *<state2>*, *<symbol2>*, *right*⟩ ∈ *I* **then**
            T := (*T* ~ {⟨*<address>*, *<symbol>*⟩}) ∪ {⟨*<address>*, *<symbol2>*⟩};
            c := ⟨*<state2>*, *<address>* + 1⟩;
    else if *c* = ⟨*<state>*, *<address>*⟩

and ⟨*<address>*, *<symbol>*⟩ ∈ *T*
and ⟨*<state>*, *<symbol>*, *<state2>*, *<symbol2>*, *stay*⟩ ∈ *I* **then**
T := (*T* ~ {⟨*<address>*, *<symbol>*⟩}) ∪ {⟨*<address>*, *<symbol2>*⟩};
else
exit;
endloop;

For the working memory representation *W* of a Turing machine *M*, the above procedure can be implemented by the following production memory.

```
lengthen_tape:
     (config)
NOT  (tape  addr:#1.head)
⇒
ADD  (tape  addr:#1.head  symbol:$)

do_left:
     (config)
     (tape  addr=#1.head)
     (instr on_state=#1.state  read=#2.symbol  move=left)
⇒
CHG  (#2,  symbol:#3.write)
CHG  (#1,  state:#3.to_state  head:#1.head-1)

do_right:
     (config)
     (tape  addr=#1.head)
     (instr on_state=#1.state  read=#2.symbol  move=right)
⇒
CHG  (#2,  symbol:#3.write)
CHG  (#1,  state:#3.to_state  head:#1.head+1)


do_stay:
     (config)
     (tape  addr=#1.head)
     (instr on_state=#1.state  read=#2.symbol  move=stay)
⇒
CHG  (#2,  symbol:#3.write)
CHG  (#1,  state:#3.to_state)
```

where every production condition corresponds exactly to an *if* condition, and every production action corresponds exactly to an *if* action. The procedural control specified by the ***loop*** and *if* structure can also be specified by the following control grammar:

LOOP → lengthen_tape LOOP
LOOP → do_left LOOP
LOOP → do_right LOOP
LOOP → do_stay LOOP
LOOP → halt_condition

where the condition of production ***halt_condition*** is satisfied when no instruction exists for the current state:

```
halt_condition:
    (config)
    (tape  addr=#1.head)
NOT (instr on_state=#1.state  read=#2.symbol)
⇒
```

This corresponds to the termination condition of the ***loop*** statement. The selection strategy used by the interpreter does not affect the interpretation, because only one production will be satisfied on any cycle. □

COROLLARY 2.1

For every Turing machine computable function there exists a control free production system that computes it.

Proof. The controlled production system defined in THEOREM 2.1 is control free, since K = {***lengthen_tape, do_left, do_right, do_stay***}* and the selection strategy does not affect the interpretation. □

This result is expected, since PPSs are control free and computationally equivalent to deterministic Turing machines.

3. LR Parsing

A definitive description of the theory of LR parsing[96] may be found in Aho and Ullman[97,98]. With the following exceptions, the terminology and notation found therein is used without explanation.

- Σ denotes an alphabet composed of lowercase letters.
- N denotes an alphabet composed of uppercase letters.
- strings in $(\Sigma \cup N)^*$ are denoted by lower case Greek letters.

Every context-free grammar (CFG) *G* defines a language L(*G*)[99,100]. A parser for *G* is a program that recognizes strings in L(*G*). One particularly useful and efficient type of parser is the LR parser. An LR parser for language L(*G*) consists of a set of language dependent LR parse tables, and the language independent LR parsing algorithm. LR(k) parsers have a number of useful properties: parse tables may be generated automatically from *G*; members of L(*G*) are recognized in linear time; and non-members of L(*G*) are recognized upon examining the first *k* incorrect symbols.

The *k* in "LR(k) parser" refers to the number of symbols of lookahead required by the parser to choose between conflicting parse actions. For any value of *k*, the set of grammars for which an LR(k) parser exists is a subset of the grammars for which an LR(k+1) parser exists. However, since the size and complexity of an LR parser is directly related to the value of *k*, most real parser generation systems generate parsers with one symbol lookahead (i.e., *k* = 1). For the purposes of this thesis, no lookahead is required and LR(0)

96 The *(k)* in *LR(k) parsing* is often omitted when the value of *k* is irrelevant.
97 Aho and Ullman, 1972, pp. 83-96.
98 Aho and Ullman, 1972, pp. 368-396.
99 Aho and Ullman, 1972, pp. 83-96.
100 Savitch, 1982, p. 10.

parsers are sufficient[101]. Descriptions, definitions and examples within this chapter refer specifically to LR(0) parsers.

3.1. LR(0) Parsing

This section presents LR(0) parsing background material for reference purposes. It is derived directly from the general LR(k) definitions of Aho and Ullman[102] for the special case, $k = 0$.

DEFINITIONS

Let G be a CFG and γ a viable prefix of G. We define $V(\gamma)$ to be the set of LR(0) items valid for γ with respect to G. We define $S = \{a \mid a = V(\gamma)$ for some viable prefix γ of $G\}$ as the *collection of sets of valid LR(0) items* for G. The collection of sets of valid LR(0) items for G', the augmented form of G, is called the *canonical collection of sets of valid LR(0) items for G*. □

For example, consider the augmented grammar

$$
\begin{array}{lllll}
(0) & S' & \rightarrow & S \\
(1) & S & \rightarrow & AaS \\
(2) & S & \rightarrow & y \\
(3) & A & \rightarrow & x
\end{array}
$$

The canonical collection of sets of LR(0) items for this grammar along with $V(\varepsilon)$ is:

$$
\begin{array}{llll}
a_0 = V(\varepsilon): & \{\ [S' & \rightarrow & \bullet S] \\
 & \ \ [S & \rightarrow & \bullet AaS] \\
 & \ \ [S & \rightarrow & \bullet y] \\
 & \ \ [A & \rightarrow & \bullet x]\ \} \\
a_1: & \{\ [S' & \rightarrow & S\bullet]\ \} \\
a_2: & \{\ [S & \rightarrow & A\bullet aS]\ \} \\
a_3: & \{\ [S & \rightarrow & y\bullet]\ \}
\end{array}
$$

---

101 Thus, all flavors of LR parsers (e.g., *LR(k)*, *LALR(k)*, *SLR(k)* for which $k \geq 0$) are also sufficient, and are collectively referred to as *LR parsers* throughout the remainder of this thesis.

102 Aho and Ullman, 1972, pp. 368-396.

a_4 :	{ [A	$\rightarrow$	$x\bullet$] }
a_5 :	{ [S	$\rightarrow$	$Aa\bullet S$]
	[S	$\rightarrow$	$\bullet AaS$]
	[S	$\rightarrow$	$\bullet y$]
	[A	$\rightarrow$	$\bullet x$] }
a_6 :	{ [S	$\rightarrow$	$AaS\bullet$] }

DEFINITION

Let $G = \langle N, \Sigma, P, S\rangle$ be a CFG and let $\mathcal{S}$ be a collection of sets of LR(0) items for G. If a is a set of LR(0) items such that $a = V(\gamma)$, where $\gamma \in (N \cup \Sigma)^*$, then $GOTO(a, X)$ is that a' such that $a' = V(\gamma X)$, where $X \in (N \cup \Sigma)$. □

DEFINITIONS

Let $G = \langle N, \Sigma, P, S\rangle$ be a CFG and let $\mathcal{S}$ be a collection of sets of LR(0) items for G. $T(a)$, the *LR(0) table associated with* the set of items a in $\mathcal{S}$, is a pair of functions $\langle f, g\rangle$. f is called the *parsing action function* and g is called the *goto function*.

1. f maps $\{\varepsilon\}$ into {*accept*, *shift*} $\cup$ {*reduce* i | i is the number of a production in P}, where
   (a) $f(\varepsilon)$ = *shift*, if $[A \rightarrow \beta_1 \bullet \beta_2]$ is in a, and $\beta_2 \neq \varepsilon$.
   (b) $f(\varepsilon)$ = *reduce* i, if $[A \rightarrow \beta\bullet]$ is in a, and $A \rightarrow \beta$ is production number i in P, $i \geq 1$.
   (c) $f(\varepsilon)$ = *accept* if $[S' \rightarrow S\ \bullet]$ is in a.
2. g, the goto function, determines the next applicable table. Some g will be applied immediately after each shift and reduction. Formally, g maps $N \cup \Sigma$ into the set of tables or the message *error*. $g(X)$ is the table associated with GOTO(a, X). If GOTO(a, X) = ∅ then $g(X)$ = *error*. □

DEFINITION

The *canonical set of LR(0) tables* for an LR(0) grammar G is the pair $\langle \mathcal{T}, T_0\rangle$ where $\mathcal{T}$ is the set of LR(0) tables associated with the canonical collection of sets of valid LR(0) items for G, and T_0 is the LR(0) table associated with $V(\varepsilon)$. □

	ε	S	A	a	b	y	x
T_0	S	T_1	T_2	–	–	T_3	T_4
T_1	A	–	–	–	–	–	–
T_2	S	–	–	T_5	–	–	–
T_3	R2	–	–	–	–	–	–
T_4	R3	–	–	–	–	–	–
T_5	S	T_6	T_2	–	–	T_3	T_4
T_6	R1	–	–	–	–	–	–

FIGURE 3.1. A Canonical Set of LR(0) Parse Tables[103]

ALGORITHM 3.1

Construction of the canonical set of LR(0) tables from a CFG.

Input. A CFG $G = \langle N, \Sigma, P, S \rangle$.

Output. The canonical set of LR(0) tables for G.

Method. Perform steps (1) through (3).

1. Construct the augmented grammar $G' = \langle N \cup \{S'\}, S, P \cup \{S' \rightarrow S\}, S' \rangle$. $S' \rightarrow S$ is to be the zeroth production.
2. From G' construct $\mathcal{S}$, the canonical collection of sets of valid LR(0) items for G[104].
3. Let $\mathcal{T}$ be the set of LR(0) tables for G, where $\mathcal{T} = \{T \mid T = T(a)$ for some $a \in \mathcal{S}\}$. Let $T_0 = T(a_0)$, where $a_0 = V(\varepsilon)$.□

For the augmented grammar above, the canonical set of LR(0) parse tables in FIGURE 3.1 would be constructed.

ALGORITHM 3.2

The LR(0) parsing algorithm.

Input. A canonical set of LR(0) parse tables $\langle \mathcal{T}, T_0 \rangle$ for an LR(0) CFG G, and an input string $z \in \Sigma^*$, which is to be parsed.

103 where, S ≡ *shift*, A ≡ *accept*, – ≡ *error*, and Rn ≡ *reduce* by production number n.

104 Definition above and Algorithm 5.8, Aho and Ullman, 1972, p. 386.

Output. If $z \in L(G)$ then output *true*; otherwise, output *false*.

Method. The state configurations for this algorithm are pairs $\langle \alpha, \chi \rangle$, where:

- α represents the parse stack (whose is on the right). Elements of α are pairs $\langle \sigma, T \rangle$, where $\sigma \in \Sigma \cup N$, and $T \in \mathcal{T}$.
- χ represents the portion of z yet to be parsed.

The initial configuration is $\langle \langle \varepsilon, T_0 \rangle, z \rangle$. At all times, let T refer to the table of the topmost pair on α. The parsing action function f of the T is applied by steps (1) through (3) until acceptance occurs or an error is encountered.

1. If $f(\varepsilon) = shift$, then perform steps (a) through (c). Otherwise, **go to** step (2).
   (a) Let $\chi = u\gamma$. Push $\langle u , g(u) \rangle$ onto the top of α, where g is the goto function of T. If $g(u) = error$, then **halt**, and output *false* [105].
   (b) Assign γ to χ; that is, remove the first symbol from χ.
   (c) **Go to** step (1).
2. If $f(\varepsilon) = reduce\ i$, where production i is of the form $A \rightarrow \beta$, then perform steps (a) through (c). Otherwise, **go to** step (3).
   (a) Remove $|\beta|$ pairs from the top of α.
   (b) Push $\langle A , g(A) \rangle$ onto the top of α, where g is the goto function of T.
   (c) **Go to** step (1).
3. If $f(\varepsilon) = accept$, then **halt** and output *true*. □

DEFINITIONS

A *parsing function* $\mathcal{F} : V_1 \times \ldots \times V_n \times \Sigma^* \rightarrow \{true, false\}$ where $V_1, \ldots, V_n$, are language dependent value sets. Given a CFG G, a *parser for* G is a function $\mathcal{F}_G : \Sigma^* \rightarrow \{true, false\}$, where $\mathcal{F}_G(z) = \mathcal{F}(v_{1,n}, z)$ and $v_{1,n} \in V_1 \times \ldots \times V_n$. □

That is, a parser for G is an instance of a language independent parsing function for which a number of language dependent arguments have been specified, usually a set of parse tables and other ancillary functions[106] used to

[105] And, in practice, transfer to the error recovery routine.
[106] cf. SEC. 3.2.

perform the parse. A parser for G determines whether a given string $z \in \Sigma^*$ is in L(G).

DEFINITION

An *LR(0) parser for a CFG G* is a predicate $\mathcal{F} : \Sigma^* \rightarrow \{true, false\}$, where $\mathcal{F}(z) = \mathcal{F}_{LR(0)}(\langle \mathcal{T}, T_0 \rangle, z)$, $\mathcal{F}_{LR(0)}$ is the LR(0) parsing predicate defined by ALGORITHM 3.2, and $\langle \mathcal{T}, T_0 \rangle$ is the canonical set of LR(0) parse tables for G. □

EXAMPLE 3.1

Consider again the context-free grammar G

$$
\begin{array}{llll}
(0) & S' & \rightarrow & S \\
(1) & S & \rightarrow & AaS \\
(2) & S & \rightarrow & y \\
(3) & A & \rightarrow & x
\end{array}
$$

represented by the canonical set of LR(0) parse tables in FIGURE 3.1. Given the input xay, a trace of the configurations generated by an LR(0) parser for G follows:

$$
\begin{array}{l}
(\langle \varepsilon, T_0 \rangle, xay) \\
(\langle \varepsilon, T_0 \rangle \langle x, T_4 \rangle, ay) \\
(\langle \varepsilon, T_0 \rangle \langle A, T_2 \rangle, ay) \\
(\langle \varepsilon, T_0 \rangle \langle A, T_2 \rangle \langle a, T_5 \rangle, y) \\
(\langle \varepsilon, T_0 \rangle \langle A, T_2 \rangle \langle a, T_5 \rangle \langle y, T_3 \rangle, \varepsilon) \\
(\langle \varepsilon, T_0 \rangle \langle A, T_2 \rangle \langle a, T_5 \rangle \langle S, T_6 \rangle, \varepsilon) \\
(\langle \varepsilon, T_0 \rangle \langle S, T_1 \rangle, \varepsilon)
\end{array}
$$

accept and output *true*. □

3.2. Enhancements to LR(0) Parsing

Currently, numerous LR parser generators[107,108] exist as part of automatic compiler generation systems. The parsers generated by these systems usually incorporate a number of features that are useful in compilers but are not part

---

107 Collins and Noonan, 1985.

108 AT & T Information Systems, pp. 350-351.

of the LR(k) parser definition. Since these extensions are essential to later parts of this thesis, a description of each feature is provided below.

3.2.1. CLR(0) Parse Tables

A CFG $G = \langle N, \Sigma, P, S \rangle$ is said to be *LR(0)* if the sets of LR(0) items generated for G are consistent. In a consistent LR(0) item set, there is exactly one value (e.g., *shift, reduce, accept*) defined for each parsing action function. In practice, however, many useful grammars are not LR(0). LR(0) parse tables for such grammars would contain multi-valued parsing action functions, where the actions returned are collectively called a "collision." For this reason, some parser generators[109,110] will produce such multi-valued functions, and allow the run-time environment to choose which action to apply. Herein, such tables will be called *CLR(0) parse tables*, where the "C" stands for "colliding." The following definitions and construction algorithms for CLR(0) tables are adaptations of the corresponding LR(0) definitions and algorithms above.

DEFINITIONS

Let $G = \langle N, \Sigma, P, S \rangle$ be a CFG, let $\mathcal{S}$ be a collection of sets of LR(0) items for G, and let $T(a) = \langle f, g \rangle$ be the LR(0) table associated with the set of items a in $\mathcal{S}$. $T_c(a)$, the *CLR(0) table associated* with the set of items a in $\mathcal{S}$, is a pair of functions $\langle f_c, g \rangle$. f_c is called the *colliding parsing action function* and g is the *goto function*.

1. f_c maps $\{\varepsilon\} \rightarrow 2^{\mathcal{A}}$, where $\mathcal{A} = \{shift, accept\} \cup \{reduce\ i \mid i$ is the number of a production, $1 \leq i \leq |P|\}$, where
   (a) $shift \in f_c(\varepsilon)$, if $[A \rightarrow \beta_1 \bullet \beta_2]$ is in a, and $\beta_2 \neq \varepsilon$.
   (b) $reduce\ i \in f_c(\varepsilon)$, if $[A \rightarrow \beta \bullet]$ is in a, and $A \rightarrow \beta$ is production number i in P, $i \geq 1$.
   (c) $accept \in f_c(\varepsilon)$, if $[S' \rightarrow S\ \bullet]$ is in a.
2. g is the goto function for normal LR(0) tables. □

Notice that $f_c(\varepsilon) = f(\varepsilon)$ when a is consistent.

---

109 Collins and Noonan, 1985.

110 AT & T Information Systems, pp. 350-351.

DEFINITION

The *canonical set of CLR(0) tables* for a CFG G is the pair $\langle \mathcal{T}_c, T_0 \rangle$, where $\mathcal{T}_c$ is the set of CLR(0) tables associated with the canonical collection of sets of LR(0) items for G, and T_0 is the CLR(0) table associated with $V(\varepsilon)$. □

For example, consider the augmented grammar

$$
\begin{array}{lll}
(0) & S' & \rightarrow S \\
(1) & S & \rightarrow a \\
(2) & S & \rightarrow ab \\
(3) & S & \rightarrow aB \\
(4) & B & \rightarrow b
\end{array}
$$

The canonical collection of sets of valid LR(0) items for this grammar along with $V(\varepsilon)$ is:

$$
\begin{array}{lll}
a_0 = V(\varepsilon): & [S' \rightarrow & \bullet S] \\
 & [S \rightarrow & \bullet a] \\
 & [S \rightarrow & \bullet ab] \\
 & [A \rightarrow & \bullet aB] \\
a_1: & [S' \rightarrow & S\bullet] \\
a_2: & [S \rightarrow & a\bullet] \\
 & [S \rightarrow & a\bullet b] \\
 & [S \rightarrow & a\bullet B] \\
 & [B \rightarrow & \bullet b] \\
a_3: & [S \rightarrow & ab\bullet] \\
 & [B \rightarrow & b\bullet] \\
a_4: & [S \rightarrow & aB\bullet]
\end{array}
$$

resulting in the canonical set of CLR(0) tables found in FIGURE 3.2.

	ε	S	B	a	b
T_0	{S}	T_1	-	T_2	-
T_1	{A}	-	-	-	-
T_2	{S, R1}	-	T_4	-	T_3
T_3	{R2, R4}	-	-	-	-
T_4	{R3}	-	-	-	-

FIGURE 3.2. Canonical CLR(0) Parse Tables[111]

ALGORITHM 3.3

Construction of the canonical set of CLR(0) tables from a CFG.

Input. A CFG $G = \langle N, \Sigma, P, S \rangle$.

Output. $\langle \mathcal{T}_c, T_0 \rangle$, the canonical set of CLR(0) tables for G.

Method. Perform steps (1) through (3).

1. Construct the augmented grammar $G' = \langle N \cup \{S'\}, S, P \cup \{S' \to S\}, S' \rangle$. $S' \to S$ is to be the zeroth production.
2. From G' construct $\mathcal{S}$, the canonical collection of sets of valid LR(0) items for G[112].
3. Let $\mathcal{T}_c$ be the set of CLR(0) tables for G; $\mathcal{T}_c = \{T \mid T = T_c(a)$ for some $a \in \mathcal{S}\}$. Let $T_0 = T_c(a_0)$, where $a_0 = V(\varepsilon)$. □

Notice that members of the range of $f_c(\varepsilon)$ are sets of parsing actions. To provide convenient access to individual members of this set the following equivalent form of CLR(0) parse tables is defined.

DEFINITION

Given $\langle \mathcal{T}_c, T_0 \rangle$, the canonical set of CLR(0) tables for a CFG G, the *alternate representation* of $\langle \mathcal{T}_c, T_0 \rangle$ is identical to $\langle \mathcal{T}_c, T_0 \rangle$ with the following exceptions:

---

111 where, S ≡ *shift*, A ≡ *accept*, - ≡ *error*, and Rn ≡ *reduce* by production number n.

112 Definition above and Algorithm 5.8, Aho and Ullman, 1972, p. 386.

2. Parse actions are ordered such that *shift* actions (of the form *shift* u) occur before *reduce* actions (of the form *reduce* i). The ordering among shift actions and the ordering among reduce actions is unspecified.
3. $f_c(i)$, where $i > 0$, denotes the ith parse action in the ordered set $f_c(\varepsilon)$. If $|f_c(\varepsilon)| < i$, then $_c(i)$ denotes *error*. □

The alternate form of the canonical CLR(0) parse tables in FIGURE 3.2 is shown in FIGURE 3.3. Henceforth, all CLR(0) tables will be represented in this alternate form.

	1	2	S	B	a	b
T_0	Sa	-	T_1	-	T_2	-
T_1	A	-	-	-	-	-
T_2	Sb	R1	-	T_4	-	T_3
T_3	R2	R4	-	-	-	-
T_4	R3	-	-	-	-	-

FIGURE 3.3. CLR(0) Parse Tables (Alternate Form)[113]

3.2.2. Disambiguation Functions

For many practical grammars it is useful to allow the parsing algorithm to choose one of many possible parsing actions at run-time based upon information collected during the parse. For example, array references and function calls are occasionally indistinguishable by an LR(1) parser for the Ada grammar. A comparison of the identifier information stored in the semantics stack and the symbol table is required to decide which reduction to make. In many LR parsing systems, these decisions are made by disambiguation functions.

[113] where, S ≡ *shift*, A ≡ *accept*, - ≡ *error*, and Rn ≡ *reduce* by production number n.

DEFINITION

A *disambiguation function* $\mathcal{d}$ for a CFG $G = \langle N, \Sigma, P, S \rangle$ is a function $\mathcal{d}: \mathcal{A} \times V_1 \times V_2 \times \ldots \times V_m \rightarrow \{true, false\} \times V_{m+1} \times V_2 \times \ldots \times V_n$ where $V_1, \ldots, V_n$ are some sets of values. $\mathcal{d}(p, v_1, v_2, \ldots, v_m) = \langle true, v_{m+1}, \ldots, v_n \rangle$ for appropriate values $v_1, v_2, \ldots, v_n$, if the parser should apply parsing action p, and $\mathcal{d}(p, v_1, v_2, \ldots, v_m) = \langle false, v_{m+1}, \ldots, v_n \rangle$, otherwise. □

This definition describes a class of application dependent functions. In general, a disambiguation function $\mathcal{d}$ takes a parse action and a list of zero or more application dependent values (i.e., $v_1, v_2, \ldots, v_m$) as arguments. Function $\mathcal{d}$ returns either *true* or *false*, depending upon whether or not the input parse action should be performed, along with a list of zero or more application dependent values (i.e., $v_{m+1}, \ldots, v_n$). The value sets $V_1, V_2, \ldots, V_n$ and the definition of a specific $\mathcal{d}$ are application dependent. Specific disambiguation functions[114] may have restrictions imposed on input and output values.

3.2.3. Semantics Function

It is often useful for an LR parser to be able to perform some side-effect operation, such as modifying a symbol table or generating object code, whenever a parsing action is performed. For this reason, some parser generator systems allow the inclusion of a *semantics function* as defined below.

DEFINITION

A *semantics function* $r : \mathcal{A} \times V_1 \times \ldots \times V_m \rightarrow V_{m+1} \times \ldots \times V_n$ where $V_1, \ldots, V_m, V_{m+1}, \ldots, V_n$ are some sets of values. □

This definition describes a class of application dependent functions. In general, a semantics function r takes a parse action and a list of zero or more application dependent values (i.e., $v_1, v_2, \ldots, v_m$) as arguments. Function r returns a list of zero or more application dependent values (i.e., $v_{m+1}, \ldots, v_n$). The value sets $V_1, V_2, \ldots, V_n$ and the definition of a specific r are application

114 cf. SECTION 4.2.1.2.

dependent. Specific semantics functions[115] may have restrictions imposed on input and output values.

3.2.4. CLR(0) Parsers

A CLR(0) parser is an LR(0) parser that incorporates the additional features described above. Similarly, the CLR(0) parsing algorithm is a modification of the standard LR(0) parsing algorithm that makes use of these additional features.

ALGORITHM 3.4

The CLR(0) parsing algorithm.

Input. A canonical set of CLR(0) parse tables $\langle \mathcal{T}_c, T_0 \rangle$ for $G = \langle N, \Sigma, P, S \rangle$, a disambiguation function $\mathcal{d}$, a semantics function r, and an input string $z \in \Sigma^*$ which is to be parsed.

Output. If $z \in L(G)$, then output *true*. Otherwise, output *false*.

Method. This algorithm is application independent and assumes that an appropriate semantics function $r : \mathcal{A} \times V_a \times \ldots \times V_b \rightarrow V_{b+1} \times \ldots \times V_c$ and disambiguation function $\mathcal{d} : \mathcal{A} \times V_d \times \ldots \times V_e \rightarrow \{true, false\} \times V_{e+1} \times \ldots \times V_h$ is available for each application, where $V_a, \ldots, V_c, V_d, \ldots, V_h$ are application dependent value sets. The state configurations for this algorithm will be pairs $\langle \alpha, \chi \rangle$, where:

- α represents the parse stack (whose top is on the right). Elements of α are triples of the form $\langle \sigma, T, j \rangle$, where $\sigma \in \Sigma \cup N$, $T \in \mathcal{T}$, and $j > 0$.
- χ represents the portion of z yet to be parsed.

The initial configuration is $(\langle \varepsilon, T_0, 1 \rangle, z)$. At all times, let T and j refer, respectively, to the table and index of the topmost triple on α. Apply the parsing action function f_c of T in steps (1) through (7) as appropriate until acceptance occurs or an error is encountered.

---

115 cf. SECTION 4.2.1.2.

1. If $f_c(j) = error$, then **go to** step (7). Otherwise **go to** step (2).
2. If $d(f_c(j), v_d, \ldots, v_e) = \langle false, , v_{e+1}, \ldots, v_h \rangle$, where $v_d, \ldots, v_h$ are values appropriate to the specific application, then **go to** step (6) below. Otherwise **go to** step (3).
3. If $f_c(j) = shift\ u$, then apply steps (a) through (d) below. Otherwise **go to** step (4).
   (a) Let $\chi = v\gamma$. If $u = v$ then push $\langle u, g(u), 1 \rangle$ onto the top of α, where g is the goto function of T. Otherwise **go to** step (7).
   (b) Assign γ to χ; that is, remove v from the front of χ.
   (c) Assign $r(shift\ u, v_a, \ldots, v_b)$ to $v_{b+1}, \ldots, v_c$, where $v_a, \ldots, v_c$ are values appropriate to the specific application.
   (d) **Go to** step (1).
4. If $f_c(j) = reduce\ i$ for some production i of the form $A \rightarrow \beta$, then apply steps (a) through (d) below. Otherwise **go to** step (5).
   (a) Assign $r(reduce\ i, v_a, \ldots, v_b)$ to $v_{b+1}, \ldots, v_c$, where $v_a, \ldots, v_c$ are values appropriate to the specific application.
   (b) Remove $|\beta|$ triples from the top of α.
   (c) Push $\langle A, g(A), 1 \rangle$ onto the top of α, where g is the goto function of T.
   (d) **Go to** step (1).
5. If $f_c(j) = accept$, then apply steps (a) and (b) below.
   (a) Assign $r(accept, v_a, \ldots, v_b)$ to $v_{b+1}, \ldots, v_c$, where $v_a, \ldots, v_c$ are values appropriate to the specific application.
   (c) **Halt**, and output *true*.
6. If $f_c(j+1) \neq error$, then replace $\langle \sigma, T, j \rangle$, the topmost triple on α, with $\langle \sigma, T, j+1 \rangle$ and **go to** step (1) above. Otherwise, **go to** step (7).
7. **Halt**, and output *false*.[116] □

DEFINITION

A *CLR(0) parser* is a predicate $\mathcal{F}: \Sigma^* \rightarrow \{true, false\}$, where $\mathcal{F}(z) = \mathcal{F}_{CLR(0)}(\langle \mathcal{T}_c, T_{c0} \rangle, d, r, z)$, $\mathcal{F}_{CLR(0)}$ is the CLR(0) parsing function defined by ALGORITHM 3.4, $\langle \mathcal{T}_c, T_{c0} \rangle$ is a canonical set of CLR(0) parse tables for some CFG G, d is a disambiguation function for G, and r is a semantics function for G. □

[116] And, in practice, transfer to the error recovery routine.

EXAMPLE 3.2

Consider again the following context-free grammar:

$$\begin{array}{lllll} (0) & S' & \rightarrow & S \\ (1) & S & \rightarrow & a \\ (2) & S & \rightarrow & ab \\ (3) & S & \rightarrow & aB \\ (4) & B & \rightarrow & b \end{array}$$

represented by the canonical set of CLR(0) parse tables in FIGURE 3.3. We define a specific disambiguation function $d : \mathcal{A} \times (\Sigma \cup N)^* \rightarrow \{true, false\}$, such that

$$d(p, \beta) = \begin{cases} false & \text{if } p = reduce\ i \wedge |\beta| \geq i \\ true & \text{otherwise;} \end{cases}$$

and a specific semantics function $r : \mathcal{A} \rightarrow \varnothing$, such that $r(p) = write(p)$, for all $p \in \mathcal{A}$. A trace of the configurations defined by ALGORITHM 3.4 for the resulting CLR(0) parser on input $z = ab$ is:

$(\langle \varepsilon, T_0, 1 \rangle, ab)$	r writes *shift a*
$(\langle \varepsilon, T_0, 1 \rangle \langle a, T_2, 1 \rangle, b)$	r writes *shift b*
$(\langle \varepsilon, T_0, 1 \rangle \langle a, T_2, 1 \rangle \langle b, T_3, 1 \rangle, \varepsilon)$	r writes *reduce* 4
$(\langle \varepsilon, T_0, 1 \rangle \langle a, T_2, 1 \rangle \langle B, T_4, 1 \rangle, \varepsilon)$	r writes *reduce* 3
$(\langle \varepsilon, T_0, 1 \rangle \langle S, T_1, 1 \rangle, \varepsilon)$	r writes *accept*
accept and output *true*.	

Note, that between the 3rd and 4th configurations, parsing action $R2$ was tried (which would result in $(\langle \varepsilon, T_0, 1 \rangle \langle S, T_1, 1 \rangle, \varepsilon)$ for the 4th configuration) and rejected because $d(reduce\ 2, ab) = false$. Also, since $d(shift\ u, \beta) = true$ for all u and β, the $R1$ action in T_2 will never be performed. □

4. The Palimpsest Parser Production System Architecture

This chapter defines a new production system architecture called a *palimpsest parser*. The organization of this chapter will resemble that of CHAPTER 2. The first section informally describes palimpsest parsers and contrasts them with conventional production system architectures. The second section addresses the structure, interpretation, and procedural control of palimpsest parsers more formally. The final section discusses the scope of this architecture. In particular, an algorithm, called the *palimpsest transformation* is presented, that transforms an arbitrary controlled production system into a palimpsest parser. It is shown that all deterministic Turing machines can be represented and interpreted by palimpsest parsers.

4.1. Introduction to Palimpsest Parsers

This section provides an informal, step-by-step description of the structure and operation of palimpsest parsers. The purpose of this section is to give the reader some understanding of basic palimpsest parser concepts before formal definitions are presented in SECTION 4.2.

4.1.1. Conflict Resolution of Controlled Production Systems

The primary purpose of conflict resolution functions in conventional production system architectures is to impose the procedural control that has been built into production systems[117]. However, conflict resolution functions are not used to impose procedural control on controlled production systems; all procedural control is explicitly specified by a control language separate

117 cf. FIGURE 1.5.

from the productions. The control language is used to impose procedural control, as in ALGORITHM 2.2, by restricting the match process, allowing only legal productions to be instantiated.

Conventional production system architectures also use conflict resolution functions to enforce *refraction*; that is, to insure that no instantiation is ever fired twice. While refraction is necessary to guard against unintentional infinite loops in uncontrolled architectures, such a rigid restriction is unnecessary for controlled production systems. First, the only loops that can occur in a controlled production system are those that are explicitly specified by the control grammar. Second, it may be desirable for the same instantiation to fire more than once; for example, on every cycle of a loop. Third, explicit implementation of refraction within the production system is trivial in those rare instances where it is necessary[118].

The above observations suggest that conflict resolution plays no useful role in the interpretation of controlled production systems, so any arbitrary conflict resolution function can be used. This suggestion is supported by COROLLARY 2.1; any Turing machine can be transformed into a production system and interpreted using any arbitrary selection strategy. For this reason, a simple conflict resolution function that allows significant performance-related optimizations is incorporated into the palimpsest parser production system architecture[119]. This conflict resolution function is called *fire first*.

4.1.2. The Fire First Conflict Resolution Function

As the name suggests, the *fire first* conflict resolution function chooses the instantiation that was entered into the conflict set first (i.e., least recently) by the match function. This conflict resolution function was chosen for use in palimpsest parsers because it significantly simplifies the interpretation algorithm for controlled production systems (i.e., ALGORITHM 2.2) as follows:

---

118 cf. SECTION 6.8.

119 For those who reject this argument, trivial modifications to the palimpsest parsers presented in this chapter allow any conventional conflict resolution function to be employed (cf. SECTION 6.8).

1. The match function may halt after finding one instantiation. There is no point in searching for other instantiations that cannot not be chosen by the conflict resolution function.
2. The conflict resolution phase can be ignored. There is no conflict to resolve if the match function returns a single instantiation.

The interpreter searches for instantiations and fires them as they are found. In effect, all conflict resolution criteria are determined by the order in which the match function searches for instantiations. For palimpsest parsers, this search order is implementation dependent.

4.1.3. Control Information

Most forward chaining production system architectures use and maintain information about a production system in order to improve performance. For example, memory support and condition membership information are maintained and used by most indexing architectures; palimpsest parsers are no exception. More importantly, palimpsest parsers also use control information, specified by a control grammar, to constrain the search for instantiations. To illustrate this directed search capability, consider a simple parsing based interpreter constructed as follows:

STEP 1: Create a copy of the control grammar, and change all terminal symbols of this grammar (i.e., production names) into identical non-terminal symbols. For the control grammar of FIGURE 4.1, the resulting grammar is:

(1) $S \rightarrow$ LOADITEM S
(2) $S \rightarrow$ OPENNEWBAY S
(3) $S \rightarrow$ STOP

STEP 2: For every non-terminal production name in this grammar, add a new grammar rule of the form $PRODUCTION_NAME \rightarrow \varepsilon$. For the controlled production system of FIGURE 4.1, the added rules are:

(4) LOADITEM $\rightarrow \varepsilon$
(5) OPENNEWBAY $\rightarrow \varepsilon$
(6) STOP $\rightarrow \varepsilon$

Production Memory

```
loaditem:
    (item in_bay=none)
    (bay  space≥#1.size  open=true)
⇒
CHG (#1, in_bay:#2.name)
CHG (#2, space:#2.space-#1.size)

opennewbay:
    (item in_bay=none)
NOT (bay  space≥#1.size open=true)
    (bay  space≥#1.size open=false)
⇒
CHG (#3, open:true)

stop:
NOT (item in_bay=none)
⇒
```

Working Memory[120]

```
a ≡ (item  name:item1  type:water  size:67  in_bay:none)
b ≡ (item  name:item5  type:scope  size:71  in_bay:bay2)
c ≡ (item  name:item7  type:book   size:53  in_bay:bay3)
d ≡ (item  name:item9  type:paper  size:34  in_bay:none)
e ≡ (item  name:item10 type:pen    size:19  in_bay:none)
f ≡ (bay   name:bay1   space:200   open:false)
g ≡ (bay   name:bay2   space:105   open:true)
```

Control Grammar

$S \rightarrow$ **loaditem** S

$S \rightarrow$ **opennewbay** S

$S \rightarrow$ **stop**

FIGURE 4.1. A Controlled Production System

[120] The letters $a, \ldots, g$ are provided to facilitate references to the corresponding working memory elements.

	1	2	3	4	S	L	O	X
T_0	R4	R5	R6	-	T_1	T_2	T_3	T_4
T_1	A	-	-	-	-	-	-	-
T_2	R4	R5	R6	-	T_5	T_2	T_3	T_4
T_3	R4	R5	R6	-	T_6	T_2	T_3	T_4
T_4	R3	-	-	-	-	-	-	-
T_5	R1	-	-	-	-	-	-	-
T_6	R2	-	-	-	-	-	-	-

FIGURE 4.2. CLR(0) Tables for FIGURE 4.1[121]

STEP 3: Generate CLR(0) tables for the resulting grammar. For the example grammar above, the tables in FIGURE 4.2 would be generated.

STEP 4: Gernerate a disambiguation function d that finds an instantiation for a given production. That is, given parsing action *reduce i,* where grammar rule number i is of the form ***PRODUCTION_NAME*** $\rightarrow \varepsilon$, and a working memory W, d will return (*true*, $w_{1,k}$) if W contains a satisfying list $w_{1,k}$ for production ***production_name,*** and (*false*, ⟨⟩), otherwise. d returns (*true*, ⟨⟩) for all other *reduce* and *accept* parse actions. The satisfying list $w_{1,k}$ is used subsequently by a semantics function. For example, given the controlled production system in FIGURE 4.1, d(*reduce* 4, W) = (*true*, ⟨a, g⟩) since the condition of production ***loaditem*** is satisfied by the satisfying list ⟨a, g⟩ from W[122]. Similarly, d(*reduce* 5, W) = (*false*, ⟨⟩).

STEP 5: Generate a semantics function r that applies the action of a given production. That is, given a parsing action *reduce i,* where grammar rule number i is of the form ***PRODUCTION_NAME*** $\rightarrow \varepsilon$, r applies the action of production ***production_name*** to a given working memory W using a satisfying list $w_{1,k}$ supplied by d, and returns the resulting working memory. For all other parsing actions r returns the original working memory. For example, given the controlled production system of FIGURE 4.1, r(*reduce* 4, ⟨a,

121 Where Ri ≡ *reduce i*, A ≡ *accept*, - ≡ *error*, L ≡ *LOADITEM*, O ≡ *OPENNEWBAY*, and X ≡ *STOP*.

122 This assumes that ⟨a, g⟩ is the first satisfying list encountered by d.

$g\rangle$, W) applies the action of ***loaditem*** by modifying the ***in_bay*** and ***space*** attribute values of a and g, respectively.

STEP 6: Construct a CLR(0) parser from the CLR(0) parse tables, disambiguation function, and the semantics function defined above.

The resulting CLR(0) parser, when supplied with the working memory from FIGURE 4.1 and an empty input string, will correctly interpret the original controlled production system as defined by ALGORITHM 2.2. The semantics function r specifies that every application of a parse action *reduce i*, where grammar rule number i is of the form ***PRODUCTION_NAME*** $\rightarrow \varepsilon$, fires an instantiation of production ***production_name***. Such an instantiation will only be fired if it is first found by the disambiguation function d. Similarly, d will only search for this instantiation if *reduce i* is the action specified by the parse tables for the current parser configuration. The parsing actions allowed in any configuration are determined by the grammar of *STEP* 2, which is derived directly from the original control grammar. In this way, the procedural control information specified by the control grammar is used to constrain the search for satisfied productions.

EXAMPLE 4.1

Consider the controlled production system of FIGURE 4.1 for which the CLR(0) parse tables are given in Figure 4.2, and the disambiguation function, and semantics function are generated by the above process. Initially, the configuration of the CLR(0) parser would be $(\langle\varepsilon, T_0, 1\rangle, \varepsilon)$ and the initial working memory would be $\{a,b,c,d,e,f,g\}$. Production ***loaditem*** , which corresponds to the current parse action *reduce 4*, has a satisfying list $\langle a, g\rangle$, so $d(reduce\ 4, \{a,b,c,d,e,f,g\}) = (true, \langle a,g\rangle)$. Applying $r(reduce\ 4, \langle a,g\rangle, \{a,b,c, d,e,f,g\})$ applies the action of ***loaditem*** resulting in a new working memory $\{a',b,c,d,e,f,g'\}$, where

a' ≡ (item **name**:item1 **type**:water **size**:67 **in_bay**:bay2)
g' ≡ (bay **name**:bay2 **space**:38 **open**:true)

and a new configuration $(\langle\varepsilon,T_0,1\rangle\langle LOADITEM,T_2,1\rangle,\varepsilon)$. Again, parsing action *reduce 4* is specified by the configuration and $d(reduce\ 4, \{a', b, c, d, e, f, g'\}) =$

(true, ⟨d, g'⟩). Applying *r(reduce 4, ⟨d, g'⟩, {a', b, c, d, e, f, g'})* results in the new working memory *{a',b,c,d',e,f,g''}*, where

d' ≡ (item **name**:item9 **type**:paper **size**:34 **in_bay**:bay2)
g'' ≡ (bay **name**:bay2 **space**:4 **open**:true)

and a new configuration $(\langle\varepsilon, T_0, 1\rangle\langle LOADITEM, T_2, 1\rangle\langle LOADITEM, T_2, 1\rangle, \varepsilon)$. Again, parsing action *reduce 4* is specified by the configuration, but no instantiation of ***loaditem*** *exists*; *d(reduce 4, {a',b,c,d',e,f,g''}) = (false, ⟨⟩)*. Configuration $(\langle\varepsilon, T_0, 1\rangle\langle LOADITEM, T_2, 1\rangle\langle LOADITEM, T_2, 2\rangle, \varepsilon)$ is entered specifying parse action *reduce 5* (i.e., production ***opennewbay***). Since *d(reduce 5, {a',b,c, d',e,f,g''}) = (true, ⟨e,f⟩)*, then *r(reduce 5, ⟨e,f⟩, {a', b, c, d', e, f, g''})* will be applied, resulting in a new working memory *{a', b, c, d', e, f', g''}* and configuration $(\langle\varepsilon,T_0,1\rangle\langle LOADITEM,T_2,1\rangle\langle LOADITEM,T_2,2\rangle\langle OPENNEWBAY,T_3,1\rangle,\varepsilon)$, where

f' ≡ (bay **name**:bay1 **space**:200 **open**:true)

In this configuration parse action *reduce 4* is again specified, and the disambiguation function *d(reduce 4, {a',b,c,d',e,f',g''}) = (true, ⟨e,–,f'⟩)*. Applying *r(reduce 4,⟨e,–,f'⟩, {a',b,c,d',e,f',g''})* results in the new working memory *{a',b,c, d',e',f'',g''}* and configuration $(\langle\varepsilon,T_0, 1\rangle\langle LOADITEM, T_2, 1\rangle\langle LOADITEM, T_2, 2\rangle \langle OPENNEWBAY,T_3,1\rangle\langle LOADITEM,T_2,1\rangle,\varepsilon)$, where

e' ≡ (item **name**:item10 **type**:pen **size**:19 **in_bay**:bay1)
f'' ≡ (bay **name**:bay1 **space**:181 **open**:true)

After unsuccessfully trying *reduce 4* and *reduce 5* the parser will be in $(\langle\varepsilon,T_0, 1\rangle\langle LOADITEM, T_2, 1\rangle\langle LOADITEM, T_2, 2\rangle\langle OPENNEWBAY, T_3, 1\rangle \langle LOADITEM,T_2,3\rangle,\varepsilon)$. Since *d(reduce 6, {a',b,c,d',e',f'',g''}) = (true, ⟨–⟩)*, application of *r(reduce 6, ⟨–⟩, {a', b, c, d', e', f'', g''})* puts the parser in a new configuration $(\langle\varepsilon,T_0, 1\rangle\langle LOADITEM, T_2, 1\rangle\langle LOADITEM, T_2, 2\rangle\langle OPENNEWBAY, T_3, 1\rangle \langle LOADITEM, T_2, 3\rangle\langle STOP, T_4, 1\rangle, \varepsilon)$ and leaves working memory unchanged. The parser will then repeatedly reduce by the three control rules until the parser reaches the configuration $(\langle\varepsilon, T_0, 1\rangle\langle S, T_1, 1\rangle, \varepsilon)$ and halts. This sample interpretation of the controlled production system of FIGURE 4.1 is exactly that defined by ALGORITHM 2.2. □

4.1.4. Memory Support Information

As described in CHAPTER 2, every *k*-predicate is composed, in part, of a one-element component that examines a single working memory element. Any evaluation of such a one-element component, such as that denoted by

```
(item  type=widget size≥66 in_bay=none)
```

on a specific working memory element, such as

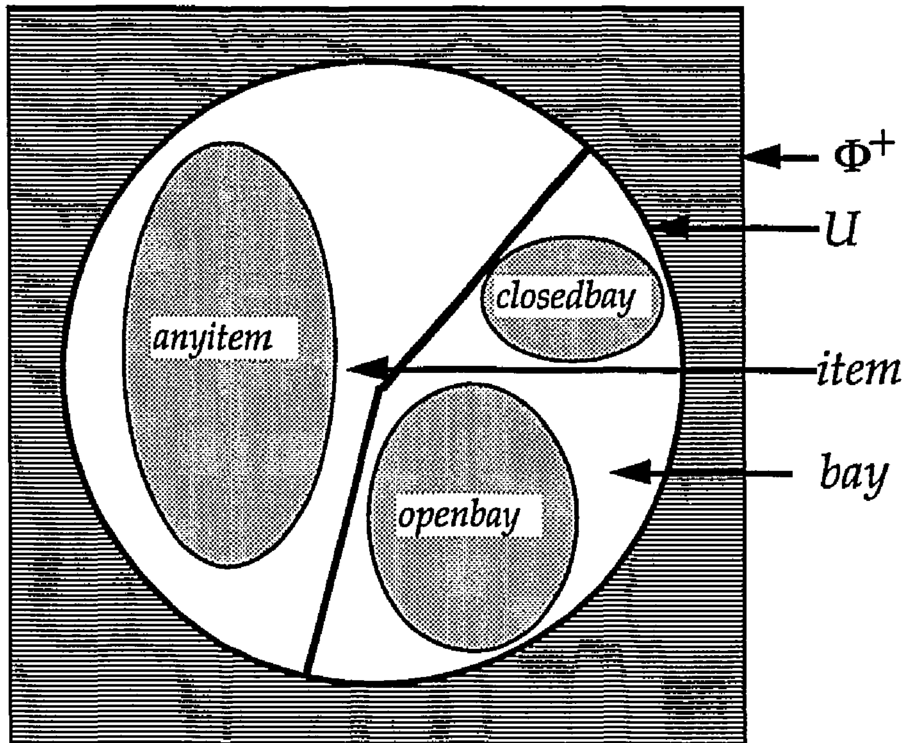
```
(item  name:w13 type:widget size:73 in_bay:none)
```

will always return the same result, in this case *true*. By remembering this result across interpretation cycles, the match process (i.e., $\mathcal{A}$ in the parsers described in the previous section) can restrict its attention to that subset of working memory that matches the one-element components of the *k*-predicates in a condition. Such information, that relates *k*-predicates to the working memory elements that partially match them, is called *memory support* information.

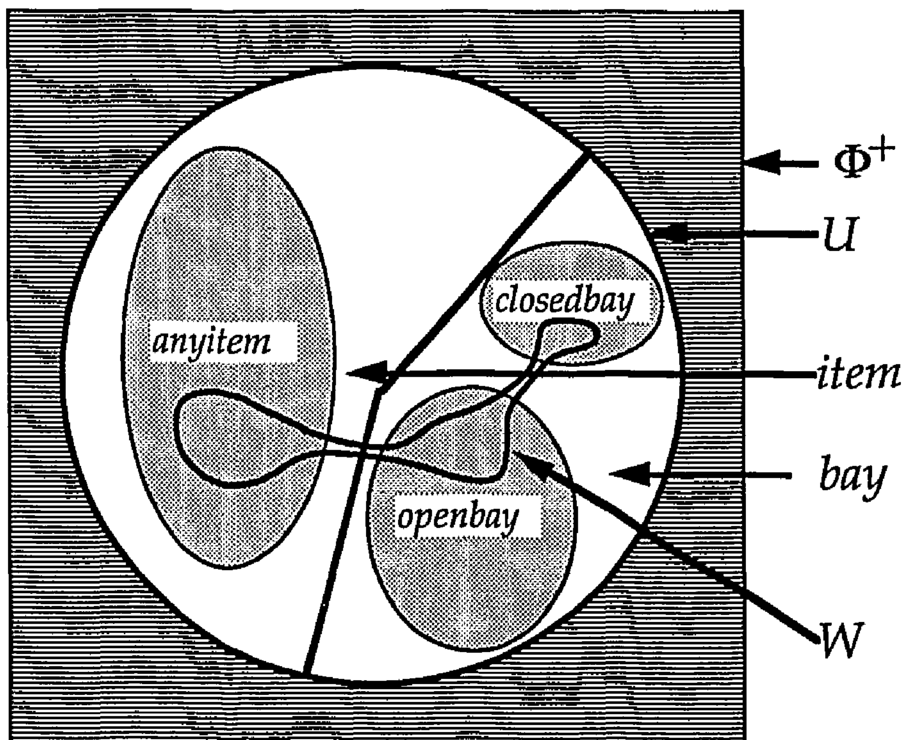
Remember from CHAPTER 2 that a working memory element universe is a set containing all working memory elements that could ever be used by a particular production system. Also, every working memory element universe is divided into distinct categories. For the production memory in FIGURE 4.1 the distinct one-element components of *k*-predicates are denoted by:

```
(item in_bay=none)
(bay  open=true)
(bay  open=false)
```

Each of these *k*-predicates is a characteristic predicate of some subset of the appropriate category, either ***item*** or ***bay***. Such a category subset will be called an *element class*. The names *ANYITEM*, *OPENBAY*, and *CLOSEDBAY* will denote the three element classes specified above, respectively. The following diagram illustrates the relationship between this particular working memory universe (*U*), the categories ***item*** and ***bay***, and the element classes *ANYITEM*, *OPENBAY*, and *CLOSEDBAY*.



Remember also that a working memory is a finite subset of a working memory element universe. This is illustrated by the next diagram:



Although a working memory (W) *can* contain working memory elements from any category (and thus from any element class), not all working memory elements are necessarily in working memory. The working memory elements that belong to an element class (e.g., E) and are also in working memory (e.g., W) are found in the intersection of that element class and that working memory (i.e., $E \cap W$).

Memory support information for a production system is available if working memory is represented as a union of sets, where each such set is the intersection of working memory with a relevant element class. The relevant element classes are defined by the distinct one-element predicate components in production memory. All working memory elements that satisfy a pattern predicate's one-element component are immediately available in the appropriate set intersection. For example, the working memory in FIGURE 4.1 can be maintained as:

$$W = \{a, d, e\} \cup \{g\} \cup \{f\}$$

where:

- $\{a, d, e\}$ is the set of elements in $ANYITEM \cap W$.
- $\{g\}$ is the set of elements in $OPENBAY \cap W$.
- $\{f\}$ is the set of elements in $CLOSEDBAY \cap W$.
- $\{b, c\}$ is the set (not shown) of elements in W that do not belong to any relevant element class; $\{b, c\} \subseteq W \sim (ANYITEM \cup OPENBAY \cup CLOSEDBAY)$.

In effect, such set intersections resemble the alpha memories of the Rete and TREAT match algorithms.

4.1.5. Condition Membership Information

Information about the satisfiability of one element components of k-predicates by working memory is called *condition membership* information. If such one-element components cannot be satisfied, then neither can the k-predicates, positive k-patterns, and conditions that contain them. A match algorithm can use such information to restrict its attention to those

conditions containing k-predicates with satisfied one-element components. Fortuitously, maintenance of memory support information in terms of element classes, as in the previous section, also provides condition membership information. The working memory elements that satisfy any one-element component of a k-predicate are immediately available in the intersection of working memory and the element class corresponding to that one-element component.

For example, consider the controlled production system of FIGURE 4.1. Production ***loaditem*** contains two positive patterns with one-element components denoted by

```
(item in_bay=none)
(bay  open=true)
```

and associated with the element classes *ANYITEM* and *OPENBAY*, respectively. For the condition of production ***loaditem*** to be satisfied it is necessary, although not sufficient, for both of the sets $ANYITEM \cap W$ and $OPENBAY \cap W$ to be non-empty. Initially, both of these sets are non-empty (i.e., $\{a, d, e\}$ and $\{g\}$, respectively) and the match algorithm should, according to condition membership, try to satisfy this condition by finding one member from each of the sets that satisfies the many-element components of the two k-predicates.

Only minor modifications to steps 2, 4, 5, and 6 of the parser generation process described in SECTION 4.1.3 are needed to generate a parser that uses condition membership information:

OLD STEP 1: Create a copy of the control grammar, and change all terminal symbols of this grammar (i.e., production names) into identical non-terminal symbols. For the control grammar of FIGURE 4.1, the resulting grammar is:

(1) $S \rightarrow$ LOADITEM S
(2) $S \rightarrow$ OPENNEWBAY S
(3) $S \rightarrow$ STOP

NEW STEP 2: For every non-terminal production name in the grammar, add a new grammar rule of the form $PRODUCTION_NAME \rightarrow x_1\, x_2\, \ldots\, x_k$,

where each x_i is a terminal symbol associated with the ith pattern in the condition of production ***production_name***, as follows:

- If the ith pattern is positive, then x_i is the element class name corresponding to the one-element component of that pattern's k-predicate. Element class name grammar symbols are in lower case to avoid confusion with the actual element class.
- Otherwise, if the ith pattern is negative, then x_i is the string ***"not_"*** concatenated with the element class name corresponding to the one-element component of that pattern's k-predicate. Element class name grammar symbols are in lower case to avoid confusion with the actual element class.

For the controlled production system of FIGURE 4.1, the rules added are:

(4)	LOADITEM	→ anyitem openbay
(5)	OPENNEWBAY	→ anyitem not_openbay closedbay
(6)	STOP	→ not_anyitem

OLD STEP 3: Generate CLR(0) tables for the resulting grammar. For the example grammar above, the tables in FIGURE 4.3 would be generated.

NEW STEP 4: Generate a disambiguation function d as follows[123]:

- For parsing actions of the form *reduce i*, where grammar rule number i is of the form ***PRODUCTION_NAME*** $\rightarrow x_1\, x_2 \ldots x_k$, and a working memory W, d will return ($true$, $w_{1,k}$) if $w_{1,k}$ is a satisfying list for production ***production_name***, and ($false$, ⟨⟩), otherwise. The satisfying list $w_{1,k}$ will be supplied as input to the semantics function r of NEW STEP (5) below.
- For parsing actions of the form *shift x*, where x corresponds to a positive pattern (i.e., $x \neq not_\alpha$ for some string α), d will return ($true$, ⟨⟩) if the element class set $X \cap W$ is non-empty, and ($false$, ⟨⟩), otherwise.
- d returns ($true$, ⟨⟩) for all other *reduce, shift,* and *accept* parsing actions.

123 It is assumed that working memory is maintained as in SECTION 4.1.4, and that d takes advantage of the resulting memory support information.

	1	2	3	S	L	B	X	a	o	c	a̲	o̲
T_0	Sa	Sa̲	-	T_1	T_4	T_5	T_6	T_2	-	-	T_3	-
T_1	A	-	-	-	-	-	-	-	-	-	-	-
T_2	So	So̲	-	-	-	-	-	-	T_7	-	-	T_8
T_3	R6	-	-	-	-	-	-	-	-	-	-	-
T_4	Sa	Sa̲	-	T_9	T_4	T_5	T_6	T_2	-	-	T_3	-
T_5	Sa	Sa̲	-	T_{10}	T_4	T_5	T_6	T_2	-	-	T_3	-
T_6	R3	-	-	-	-	-	-	-	-	-	-	-
T_7	R4	-	-	-	-	-	-	-	-	-	-	-
T_8	Sc	-	-	-	-	-	-	-	-	T_{11}	-	-
T_9	R1	-	-	-	-	-	-	-	-	-	-	-
T_{10}	R2	-	-	-	-	-	-	-	-	-	-	-
T_{11}	R5	-	-	-	-	-	-	-	-	-	-	-

FIGURE 4.3. New CLR(0) Parse Tables for FIGURE 4.1[124]

For example, given the controlled production system in FIGURE 4.1, *d*(*reduce 4, W*) = (*true*, ⟨*a, g*⟩) since the condition of production ***loaditem*** is satisfied by the satisfying list ⟨*a, g*⟩ from *W*. Similarly, *d*(*reduce 5, W*) = (*false*, ⟨⟩). Also, *d*(*shift anyitem, W*) initially returns (*true*, ⟨⟩) since the set *ANYITEM* ∩ W = {*a, d, e*} is non-empty.

NEW STEP 5: Generate a semantics function *r* that applies the action of a given production[125]. That is, given a parsing action *reduce i,* where grammar rule number *i* is of the form ***PRODUCTION_NAME*** $\rightarrow x_1\, x_2\, \ldots\, x_k$, *r* applies the action of production ***production_name*** to a given working memory *W* using a satisfying list $w_{1,k}$ supplied by *d* of NEW STEP (4) above, and returns the resulting working memory. For all other parsing actions *r* returns the

124 Where *Su* ≡ *shift u*, R*i* ≡ *reduce i*, A ≡ *accept*, - ≡ *error*, L ≡ *LOADITEM*, X ≡ *STOP*, B ≡ *OPENNEWBAY*, a ≡ *anyitem*, o ≡ *openbay*, c ≡ *closedbay*, a̲ ≡ *not_anyitem*, and o̲ ≡ *not_openbay*.

125 It is assumed that the application of add operations by function *r* maintains the working memory structure as in SECTION 4.1.4.

original working memory. For example, given the controlled production system of FIGURE 4.1, *r(reduce 4,* $\langle a, g \rangle$*, W)* applies the action of ***loaditem*** by modifying the ***in_bay*** and ***space*** attribute values of *a* and *g*, respectively.

NEW STEP 6: Construct a modified CLR(0) parser from the CLR(0) tables, disambiguation function, and semantics function defined above. This parser differs from a normal CLR(0) parser in two important ways. First, the input to the parser is a working memory, not a string, so parse actions of the form *shift u* cannot be applied in the normal CLR(0) parsing sense. The parser should put the appropriate triple onto the parse stack, but should ignore the remaining input, since there is none. Although it might a good idea to use some symbol instead of *shift* for these parse actions, the present use of parse tables created by existing parser generators dictates the name of this parse action.

Second, the parsing algorithm must backtrack over failed shift actions whenever the parser becomes blocked. This allows all actions for a given configuration to be tried, and thus allows all legal productions to be examined before the parser halts.

The resulting parser, called a *palimpsest parser*, uses control information to fire productions in the same sequence as the parsers of SECTION 4.1.3. The only differences are:

- Memory support information, as described in SECTION 4.1.4, is used to speed up the match processing in *d*, and
- Condition membership information is used to short circuit the costly match processing of unsatisfiable conditions.

The limited backtracking of the palimpsest parsing algorithm is necessary because a normal CLR(0) parser halts and signals an error when the disambiguation function disallows all legal parsing actions in the current configuration. For example, in table T_5, after shifting ***openbay***, the disambiguation function may discover that the many-element components of the patterns in ***loaditem*** cannot be satisfied by *W*, and the only parsing action in T_5, a reduction by *"LOADITEM → anyitem openbay,"* cannot be applied. In this situation, the palimpsest parsing algorithm can "unshift" terminal

symbols and try other parsing actions in previous configurations. Notice that backtracking over reduce actions is proscribed, as that would involve the costly "unfiring" of production actions.

EXAMPLE 4.2

The following is a trace of the palimpsest parser defined above for the controlled production system in FIGURE 4.1. This working memory will be represented as a union of sets, where the elements of each set denote working memory elements as described above. Since the input string of a palimpsest parser is always ε, it is ignored and replaced in configurations by the current working memory W; initially $\{a, d, e\} \cup \{g\} \cup \{f\} \cup \{b, c\}$[126].

$(\langle \varepsilon, T_0, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle anyitem, T_2, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle anyitem, T_2, 1\rangle\langle openbay, T_7, 1\rangle, W)$

A reduction by rule (4) (i.e., ***loaditem***), is applied, changing

$a \equiv$ (item **name**:item1 **type**:water **size**:67 **in_bay**:none)
$g \equiv$ (bay **name**:bay2 **space**:105 **open**:true)

into

$a' \equiv$ (item **name**:item1 **type**:water **size**:67 **in_bay**:bay2)
$g' \equiv$ (bay **name**:bay2 **space**:38 **open**:true)

and W becomes $\{d, e\} \cup \{g'\} \cup \{f\} \cup \{a', b, c\}$.

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 1\rangle\langle openbay, T_7, 1\rangle, W)$

A reduction by rule (4) (i.e., ***loaditem***), is applied again, changing

$d \equiv$ (item **name**:item9 **type**:paper **size**:34 **in_bay**:none)
$g' \equiv$ (bay **name**:bay2 **space**:38 **open**:true)

into

$d' \equiv$ (item **name**:item9 **type**:paper **size**:34 **in_bay**:bay2)
$g'' \equiv$ (bay **name**:bay2 **space**:4 **open**:true)

and W becomes $\{e\} \cup \{g''\} \cup \{f\} \cup \{d', a', b, c\}$.

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle, W)$

126 Throughout this example, the last sets in the representations of W (e.g., $\{b, c\}$) are displayed, although the working memory elements in this set will never be used.

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 1\rangle$
$\langle openbay, T_7, 1\rangle, W)$

d' disallows the only action *reduce* 4 in T_7 , so backtrack.

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 2\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 2\rangle$
$\langle not_anyitem, T_8, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle anyitem, T_2, 2\rangle$
$\langle not_anyitem, T_8, 1\rangle\langle closedbay, T_{11}, 1\rangle, W)$

A reduction by rule (5) (i.e., ***opennewbay***), is applied, changing

f ≡ (bay **name**:bay1 **space**:200 **open**:false)

into

f' ≡ (bay **name**:bay1 **space**:200 **open**:true)

and W becomes $\{e\} \cup \{f', g''\} \cup \varnothing \cup \{d', a', b, c\}$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle,$
$W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle anyitem, T_2, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle anyitem, T_2, 1\rangle\langle openbay, T_7, 1\rangle, W)$

A reduction by rule (4) (i.e., ***loaditem***), is applied again, changing

e ≡ (item **name**:item10 **type**:pen **size**:19 **in_bay**:none)
f' ≡ (bay **name**:bay1 **space**:200 **open**:true)

into

e' ≡ (item **name**:item10 **type**:pen **size**:19 **in_bay**:bay1)
f'' ≡ (bay **name**:bay1 **space**:181 **open**:true)

and W becomes $\varnothing \cup \{f'', g''\} \cup \varnothing \cup \{e', d', a', b, c\}$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle LOADITEM, T_4, 1\rangle, W)$

d' disallows the first action in $T_4$1, so increment j

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle LOADITEM, T_4, 2\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle LOADITEM, T_4, 2\rangle\langle not_anyitem, T_3, 1\rangle, W)$

A reduction by rule (6) does not change working memory

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle LOADITEM, T_4, 2\rangle\langle STOP, T_6, 1\rangle, W)$

A reduction by control rule (3) does not change working memory

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle LOADITEM, T_4, 2\rangle\langle S, T_9, 1\rangle, W)$

Reductions by control rules (1) and (2) do not change working memory

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle OPENNEWBAY, T_5, 1\rangle$
$\langle S, T_{10}, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle S, T_9, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle LOADITEM, T_4, 1\rangle\langle S, T_9, 1\rangle, W)$

$(\langle \varepsilon, T_0, 1\rangle\langle S, T_1, 1\rangle, W)$

Accept and output $W = \{f'', g'', e', d', a', b, c\}$. □

4.2. Palimpsest Parser Definitions and Theorems

4.2.1. Structure

In the previous section, palimpsest parsers and related concepts were described informally. This section presents the analagous definitions and theorems.

4.2.1.1. Working Memory

First, the definitions related to working memory are repeated for review.

DEFINITIONS

For a given alphabet Φ, $U \subseteq \Phi^+$ is a *working memory element universe,* and elements of U are called *working memory elements.* Each U is partitioned into finitely many disjoint subsets $C^{(1)}, \ldots, C^{(n)}$, called *categories.* Each category $C \in \{C^{(1)}, \ldots, C^{(n)}\}$ is associated with a finite number m of maps $A_1, \ldots, A_m$, called *attributes* that map C into sets $V_1, \ldots, V_m$, of *values;* that is, $A_i : C \rightarrow V_i$ for $i \in \{1, \ldots, m\}$, where m, the A_i's, and V_i's all depend on C. A finite set of $W \subseteq U$ is called a working memory. □

To facilitate the collection and maintenance of memory support and condition membership information the concept of an element class in introduced.

DEFINITIONS

An *element class* is a subset of a category. For any element class $E \subseteq C$ the one-element 1-predicate $p_1^1 : C \rightarrow \{true, false\}$ that is the characteristic function of E is called the *element class specification* of *e*. □

That is, for every one-element 1-predicate p_1^1 defined on C there exists an element class $E = \{w \in C \mid p_1^1(w)\}$. Also, for every k-predicate p_i^k, the corresponding $p_i^k{\downarrow}$ specifies an element class. If $p_i^k{\downarrow}$ is a tautology, as in a vacuous or many-element k-predicate, the element class defined is equivalent to the entire category C. Henceforth, the superscripts and subscripts on a one-element 1-predicate, like p_1^1, will be omitted when p is known to be an element class specification.

DEFINITIONS

An *element class table entry* is a pair $\langle id, p\rangle$, where *id* denotes an arbitrary symbol, and p is an element class specification. A set of element class table entries $\mathcal{E} = \{\langle id_1, p_1\rangle, \langle id_2, p_2\rangle, \ldots, \langle id_m, p_m\rangle\}$ is called an *element class table* if all entries have unique *id*'s. If for production memory $\mathcal{P}$ and element class table $\mathcal{E}$, $\forall\langle n, c, a\rangle \in \mathcal{P}, \forall\langle s_i, p_i^k\rangle \in c, \exists\langle id, p_i^k{\downarrow}\rangle \in \mathcal{E}$, then $\mathcal{E}$ is said to be *derived from* $\mathcal{P}$. □

For example, the following element class table is derived from the production memory in FIGURE 4.1:

$$\{ \langle ANYITEM, \ (w \in item,\ in_bay(w) = none)\rangle,\ \langle OPENBAY, \ (w \in bay,\ open(w) = true)\rangle,\ \langle CLOSEDBAY, \ (w \in bay,\ open(w) = false)\rangle \}$$

4.2.1.2. Production Memory

DEFINITION

A *production memory grammar* for a production memory $\mathcal{P}$ and element class table $\mathcal{E}$ derived from $\mathcal{P}$ is a CFG $G_{\mathcal{P}} = \langle N_{\mathcal{P}}, \Sigma_{\mathcal{P}}, P_{\mathcal{P}}, S_{\mathcal{P}}\rangle$, where:

1. $N_{\mathcal{P}} = N(\mathcal{P})$.
2. $\Sigma_{\mathcal{P}} = \{id \mid \langle id, p\rangle \in \mathcal{E}\} \cup \{not_id \mid \langle id, p\rangle \in \mathcal{E}\}$.
3. $|\mathcal{P}| = |P_{\mathcal{P}}|$, and for every production $\langle n, c, a\rangle$ in $\mathcal{P}$ there is a corresponding rule of the form $n \rightarrow x_1\ x_2 \ldots x_k$ in $P_{\mathcal{P}}$ such that
   (a) $\langle +, p_i^k\rangle \in c$ implies that $x_i = id$ and $\langle id, p_i^k{\downarrow}\rangle \in \mathcal{E}$.
   (b) $\langle -, p_i^k\rangle \in c$ implies that $x_i = \text{not_}id$ and $\langle id, p_i^k{\downarrow}\rangle \in \mathcal{E}$.
4. $S_{\mathcal{P}}$ is any element of $N_{\mathcal{P}}$. □

That is, a production memory grammar is a CFG with exactly one rule for every production in a production memory that has been constructed as in *NEW STEP 2* of SECTION 4.1.5. The production memory grammar for the production memory in FIGURE 4.1 is

LOADITEM → anyitem openbay
OPENNEWBAY → anyitem not_openbay closedbay
STOP → not_anyitem

For the following definition, remember from SECTION 2.2.3.1 that a control language grammar G_K for a production memory $\mathcal{P}$ is defined on the alphabet $N(\mathcal{P})$.

DEFINITIONS

Let $G_K = \langle N_K, \Sigma_K, P_K, S_K\rangle$ be a control grammar and $G_{\mathcal{P}} = \langle N_{\mathcal{P}}, \Sigma_{\mathcal{P}}, P_{\mathcal{P}}, S_{\mathcal{P}}\rangle$ be a production memory grammar for $\mathcal{P}$. A *palimpsest grammar* for $\mathcal{P}$ and K

is a CFG $G_{\mathcal{P}K} = \langle N_K \cup N_{\mathcal{P}}, \Sigma_{\mathcal{P}}, P_K \cup P_{\mathcal{P}}, S_K \rangle = \langle N_{\mathcal{P}K}, \Sigma_{\mathcal{P}K}, P_{\mathcal{P}K}, S_{\mathcal{P}K} \rangle$. G_K is called the *control component* of $G_{\mathcal{P}K}$, and $G_{\mathcal{P}}$ is called the *production memory component* of $G_{\mathcal{P}K}$. □

That is, a palimpsest grammar for $\mathcal{P}$ is a combination of a production memory grammar for $\mathcal{P}$ and a control grammar for $\mathcal{P}$ that has been constructed as in *OLD STEP 1* and *NEW STEP 2* of SECTION 4.1.5. The palimpsest grammar for the controlled production system of FIGURE 4.1 is

S	→ LOADITEM *S*
S	→ OPENNEWBAY *S*
S	→ STOP
LOADITEM	→ anyitem openbay
OPENNEWBAY	→ anyitem not_openbay closedbay
STOP	→ not_anyitem

Notice that every rule in the production memory component of a palimpsest grammar represents a production in the original production memory; similarly, every terminal symbol represents a pattern. In order to make such correspondences explicit in later definitions, the following functions are defined.

DEFINITIONS

Let $G_{\mathcal{P}K}$ be a palimpsest grammar. The function $PROD_{\mathcal{P}K}$ maps $\{1, \ldots, |P_{\mathcal{P}K}|\} \rightarrow \mathcal{P} \cup \{error\}$ such that:

- If the ith rule in $G_{\mathcal{P}K}$ is in the production memory component, then $PROD_{\mathcal{P}K}(i) = \langle n, c, a \rangle$, the production in $\mathcal{P}$ corresponding to that rule.
- Otherwise, $PROD_{\mathcal{P}K}(i) = error$.

Let $\mathcal{E}$ be an element class table derived from $\mathcal{P}$. The function $SIGN(id)$ maps $\Sigma_{\mathcal{P}K} \rightarrow \{+, -\}$ such that:

- $SIGN(x) = -$, if x = not_*id* and there exists an $\langle id, p \rangle \in \mathcal{E}$.
- $SIGN(x) = +$, otherwise.

Similarly, the function $EC_{\mathcal{P}K}$ maps $\Sigma_{\mathcal{P}K} \rightarrow 2^U$ such that

- If x is of the form not_*id*, then $EC_{\mathcal{PK}}(x)$ returns the element class denoted by $\langle id, p\rangle \in \mathcal{E}$, that is $\{w \mid w \in C \text{ and } p(w) = true\}$, where $p : C \rightarrow \{true, false\}$.
- Otherwise, $EC_{\mathcal{PK}}(x)$ returns the element class denoted by $\langle x, p\rangle \in \mathcal{E}$. □

DEFINITION

The *palimpsest disambiguation function* for a palimpsest grammar $G_{\mathcal{PK}}$ and production memory $\mathcal{P}$ is a disambiguation function d that maps $\mathcal{A} \times 2^{U} \rightarrow \{true, false\} \times U^*$, where $\mathcal{A}$ is the set of parsing actions in the canonical set of CLR(0) parse tables for $G_{\mathcal{PK}}$, such that, for all $W \subseteq U$:

- $d(shift\ x,\ W) = (false, \langle\rangle)$, if $SIGN(x) = +$ and $EC_{\mathcal{PK}}(x) \cap W = \varnothing$.
- $d(reduce\ i,\ W) = (false, \langle\rangle)$, if $PROD_{\mathcal{PK}}(i) = \langle n, c, a\rangle$, and c cannot be satisfied by W.
- $d(reduce\ i,\ W) = \langle true, w_{1,k}\rangle$, if $PROD_{\mathcal{PK}}(i) = \langle n, c, a\rangle$, and $\forall\langle +, p_i^k\rangle \in c, \exists w_{1,k} \in W^k$ such that $p_i^k(w_{1,k}) = true$, and $\forall\langle -, p_i^k\rangle \in c, \forall w_i \in C_i, p_i^k(w_{1,k}) = false$.
- $d(p,\ W) = (true, \langle\rangle)$, otherwise, for all other parse actions $p \in \mathcal{A}$. □

As described in SECTION 4.1, the purpose of d is to find production instantiations. When applied to shift actions, d uses condition membership information to short circuit unsatisfiable productions. Similarly, when applied to reduce actions, d tests the many-element components of k-predicates to find a satisfying list from among those subsets of working memory defined by memory support information. For example, given the CLR(0) table in FIGURE 4.1, the palimpsest disambiguation function for the controlled production system of FIGURE 4.1 would be

$$
\begin{aligned}
d(shift\ \mathbf{anyitem},\ W) &= [(ANYITEM \cap W \neq \varnothing), \langle\rangle] \\
d(shift\ \mathbf{openbay},\ W) &= [(OPENBAY \cap W \neq \varnothing), \langle\rangle] \\
d(shift\ \mathbf{closedbay},\ W) &= [(CLOSEDBAY \cap W \neq \varnothing), \langle\rangle] \\
d(shift\ \mathbf{not_anyitem},\ W) &= (true, \langle\rangle) \\
d(shift\ \mathbf{not_openbay},\ W) &= (true, \langle\rangle) \\
d(reduce\ 4,\ W) &= \langle true, w_{1,2}\rangle \text{ if } \exists w_1, w_2 \in (ANYITEM \cap W) \times \\
&\quad (OPENBAY \cap W),\ space(w_2) \geq size(w_1).
\end{aligned}
$$

$d(reduce\ 4, W)$	$=$ $(false, \langle\rangle)$, otherwise.
$d(reduce\ 5, W)$	$=$ $(true, \langle w_1, w_3\rangle)$, if $\exists w_1, w_3 \in (ANYITEM \cap W) \times (CLOSEDBAY \cap W), \forall w_2 \in (OPENBAY \cap W), space(w_3) \geq size(w_1) \wedge space(w_2) < size(w_1)$.
$d(reduce\ 5, W)$	$=$ $(false, \langle\rangle)$, otherwise.
$d(reduce\ 6, W)$	$=$ $(true, \langle\rangle)$.
$d(reduce\ 1, W)$	$=$ $(true, \langle\rangle)$.
$d(reduce\ 2, W)$	$=$ $(true, \langle\rangle)$.
$d(reduce\ 3, W)$	$=$ $(true, \langle\rangle)$.
$d(accept, W)$	$=$ $(true, \langle\rangle)$.

Notice that $d(shift\ anyitem, W) = [(ANYITEM \cap W \neq \emptyset), \langle\rangle]$ represents a use of condition membership information; d will not allow the shift because W contains no elements that will satisfy the corresponding pattern. Similarly, $d(reduce\ 4, W) = \langle true, w_{1,2}\rangle$ *if* $\exists w_1, w_2 \in (ANYITEM \cap W) \times \ldots$ represents a use of memory support information; many-element predicate components only need to be evaluated on working memory elements that already satisfy the one-element predicate components.

DEFINITION

The *palimpsest semantics function* for a palimpsest grammar $G_{\mathcal{P}K}$ and a production memory $\mathcal{P}$ is a semantics function $r : \mathcal{A} \times U^* \times 2^U \rightarrow 2^U$, where $\mathcal{A}$ is the set of parsing actions in the canonical set of CLR(0) tables for $G_{\mathcal{P}K}$, such that, for all $W \subseteq U$:

- $r(reduce\ i, w_{1,k}, W) = a(w_{1,k}, W)$, if $PROD_{\mathcal{P}K}(i) = \langle n, c, a\rangle$.
- $r(d, w_{1,k}, W) = W$, for all other parse actions $p \in \mathcal{A}$. □

In other words, if rule number i is part of the production memory component of $G_{\mathcal{P}K}$, then a palimpsest reduce function applies the action of the corresponding production to working memory; otherwise, a palimpsest reduce function does nothing. For example, given the CLR(0) table in FIGURE 4.1, the palimpsest semantics function for the controlled production system of FIGURE 4.1 would be

$r(reduce\ 4, \langle w_1, w_2\rangle, W)$	=	$(W \sim \{w_1, w_2\}) \cup \{w_a, w_b\}$, where $w_a \in item$, $name(w_a) = name(w_1)$, $type(w_a) = type(w_1)$, $size(w_a) = size(w_1)$, $bay(w_a) = name(w_2)$, and $w_b \in bay$, $name(w_b) = name(w_2)$, $space(w_b) = (space(w_2) - size(w_1))$, $open(w_b) = open(w_2)$.
$r(reduce\ 5, \langle w_1, w_3\rangle, W)$	=	$(W \sim \{w_3\}) \cup \{w_a\}$, where $w_a \in bay$, $name(w_a) = name(w_3)$, $space(w_a) = space(w_3)$, $open(w_a) = true$.
$r(reduce\ 6, \langle\rangle, W)$	=	W.
$r(reduce\ 1, \langle\rangle, W)$	=	W.
$r(reduce\ 2, \langle\rangle, W)$	=	W.
$r(reduce\ 3, \langle\rangle, W)$	=	W.
$r(shift\ \mathbf{anyitem}, \langle\rangle, W)$	=	W.
$r(shift\ \mathbf{openbay}, \langle\rangle, W)$	=	W.
$r(shift\ \mathbf{closedbay}, \langle\rangle, W)$	=	W.
$r(shift\ \mathbf{not_anyitem}, \langle\rangle, W)$	=	W.
$r(shift\ \mathbf{not_closedbay}, \langle\rangle, W)$	=	W.
$r(accept, \langle\rangle, W)$	=	W.

It is assumed that, when adding elements to working memory (e.g., $W \sim \{w_1, w_2\}) \cup \{w_a, w_b\}$), r maintains working memory as a union of element class sets as described in SECTION 4.1.4.

The following definition ties all previous definitions in this section together to define the production system architecture of this thesis: the palimpsest parser.

DEFINITION

A *palimpsest parser* for $\mathcal{P}$ and K is a function $\mathcal{F} : 2^U \rightarrow 2^U$ of the form $\mathcal{F}(W) = \mathcal{F}_P(\langle \mathcal{T}_c, T_0\rangle, d, r, W)$, where:

- $\mathcal{F}_P$ is the palimpsest parsing algorithm (ALGORITHM 4.1, below),

- $\langle \mathcal{T}_c, T_0 \rangle$ is the canonical set of CLR(0) parse tables, generated from a palimpsest grammar $G_{\mathcal{P}K}$ for a production memory $\mathcal{P}$ and a control language K of $\mathcal{P}$.
- $\mathcal{d}$ is a palimpsest disambiguation function for $G_{\mathcal{P}K}$ and $\mathcal{P}$.
- r is a palimpsest semantics function for $G_{\mathcal{P}K}$ and $\mathcal{P}$. □

Examples of palimpsest parsers can be found in SECTION 4.1.5 and Appendix B.

4.2.2. Interpretation

As described in SECTION 4.1.5, the palimpsest parsing algorithm is a modified CLR(0) parsing algorithm that backtracks over shift actions when an error is encountered. The input string to a palimpsest parser is always empty, and is ignored. In its place, the input to the palimpsest parsing algorithm is a working memory.

ALGORITHM 4.1

The palimpsest parsing algorithm.

Input. A canonical set of CLR(0) parse tables $\langle \mathcal{T}_c, T_0 \rangle$, a palimpsest disambiguation function $\mathcal{d}$, a palimpsest semantics function r, and an input working memory W.

Output. A modified working memory.

Method. The state configurations will be pairs $\langle \alpha, \chi \rangle$, where:

- α represents the parse stack (whose top is on the right). Elements of α are triples of the form $\langle \sigma, T, j \rangle$, where $\sigma \in \Sigma \cup N$, $T \in \mathcal{T}$, and $j > 0$.
- χ represents current contents of working memory.

The initial configuration is $(\langle \varepsilon, T_0, 1 \rangle, z)$. At all times, let σ, T, and j refer, respectively, to the symbol, table, and index of the topmost triple on α. Apply the parsing action function f_c of T in steps (1) through (8) as appropriate until acceptance occurs or an error is encountered.

1. If $f_c(j)$ = *error*, then **go to** step (7). Otherwise **go to** step (2).
2. If $d(f_c(j), \chi) = (false, \langle\rangle)$, then **go to** step (6). Otherwise **go to** step (3).
3. If $f_c(j)$ = *shift u*, then apply (a) through (c) below, as appropriate. Otherwise, **go to** step (4).
   (a) Push $\langle u, g(u), 1\rangle$ onto the top of α, where g is the goto function of T. Otherwise **go to** step (7).
   (b) Assign $r(shift\ u, \langle\rangle, \chi)$ to χ.
   (c) **Go to** step (1).
4. If $f_c(j)$ = *reduce i* for some production i of the form $A \rightarrow \beta$, then apply steps (a) through (d) below. Otherwise **go to** step(5).
   (a) Assign $r(reduce\ i, w_{1,k}, \chi)$ to χ, where $w_{1,k}$ is the list of working memory elements in $d(reduce\ i, \chi) = (true, w_{1,k})$ from step (2).
   (b) Remove $|\beta|$ triples from the top of α.
   (c) Push $\langle A, g(A), 1\rangle$ onto the top of α, where g is the goto function of T.
   (d) **Go to** step (1).
5. If $f_c(j)$ = *accept*, then apply steps (a) and (b) below.
   (a) Assign $r(accept, \langle\rangle, \chi)$ to χ.
   (b) **Halt**, and output χ.
6. If$f_c(j+1) \neq error$, then replace $\langle\sigma, T, j\rangle$ the topmost triple on α with $\langle\sigma, T, j+1\rangle$ and **go to** step (1) above. Otherwise, **go to** step (7).
7. If $\sigma \notin \Sigma_{\mathcal{PK}}$ or $\sigma = \varepsilon$, then **go to** step (8). Otherwise,
   (a) Remove the topmost triple from α.
   (b) Increment j.
   (c) **Go to** step (1).
8. **Halt**, and output χ. □

Notice that steps (3b) and (5a) apply the semantics function for *shift* and *accept* parsing actions; but, according to the definition, the semantics function does nothing for these parsing actions. These two steps are included in the algorithm because the production system programmer may want to augment a palimpsest parser with additional semantics between production firings and upon termination. EXAMPLE 4.2 in SECTION 4.1.5 provides a trace of the palimpsest parsing algorithm for the controlled production system in FIGURE 4.1.

4.2.3. Procedural Control

As described in SECTION 4.1.3, only legal instantiations are fired by a palimpsest parser. Unfortunately, a palimpsest parser is not *guaranteed* to find a legal production when one exists unless the control grammar is LR(0). For example, consider the following LR(1) control grammar:

(1)	MODULE_47	$\rightarrow$ PATH_A p_1
(2)	MODULE_47	$\rightarrow p_2$ PATH_B p_3
(3)	PATH_A	$\rightarrow p_2\ p_4$
(4)	PATH_B	$\rightarrow p_4$

After firing productions p_2 and p_4 a palimpsest parser will be in a configuration in which reductions by both rule (3) and rule (4) are applicable. The palimpsest parser will reduce by rule (3), and then attempt to instantiate p_1. If p_1 cannot be instantiated the parser will erroneously halt. Productions p_1 and p_3 are both legal, yet no attempt will be made to instantiate p_3. The problem lies in the fact that ALGORITHM 4.1 cannot backtrack over reductions (i.e., non-terminals). The reduction by rule (3) pops two triples from the parse stack, each containing a symbol, a table, and an index. The information that p_3 is a legal production is lost.

There are a number of possible solutions to this problem. First, prior to the generation of a palimpsest parser, it may be possible to transform a control into an equivalent LR(0) grammar. For example, the following LR(0) control grammar generates the same language as the the LR(1) control grammar above

(1)	OOPS	$\rightarrow p_2\ p_4\ p_1$
(2)	OOPS	$\rightarrow p_2\ p_4\ p_3$

At present all such transformations are performed manually. To date, all transformation attempts have been successful[127]. Second, a production system language based upon the palimpsest parser architecture can be restricted to allow only LR(0) control grammars. Third, non-LR(0) control

[127] Approximately 10 such transformations have been performed. Plans for future research include the automatiing as much of the tranformation process as may be possible, at least for the more common cases.

grammars can be flagged at palimpsest parser generation time, forcing the production system programmer to fix the control grammar. Fourth, ALGORITHM 4.1 can be easily modified to perform full backtracking; however, such modification would detrimentally affect the time and space costs of the algorithm. Full backtracking is probably not justified considering the rarity of non-LR(0) control grammars in practice.

The very nature of a modular production system language makes non-LR(0) control grammars unlikely. By definition, each module is designed to accomplish a specific task or goal by applying one of several different sequences of lower level productions or production modules. It seems unlikely, that a single module would specify many different ways of firing a sequence of productions, when one of the primary goals of the modular approach is to avoid such duplication.

4.3. Scope of Palimpsest Parsers

This section presents the two primary theorems of the thesis: that every production system can be transformed into a *palimpsest parser* (THEOREM 4.1), and that for every deterministic Turing machine there exists an equivalent palimpsest parser (THEOREM 4.2).

LEMMA 4.1

For every production memory $\mathcal{P}$ there exists an element class table $\mathcal{E}$ derived from $\mathcal{P}$.

Proof. By construction (ALGORITHM 4.2 below). □

ALGORITHM 4.2

Derivation of an element class table from a production memory.

Input. A production memory $\mathcal{P}$.

Output. An element class table $\mathcal{E}$ derived from $\mathcal{P}$.

Method. Initialize $\mathcal{E}$ to $\varnothing$. For all productions $\langle n, c, a \rangle$ in $\mathcal{P}$ perform steps (1) and (2) below.

1. If $\exists\langle s_i, p_i^k\rangle \in c$ such that $\forall\langle id, p\rangle \in \mathcal{E}, p_i^k\downarrow \neq p$, then assign $p_i^k\downarrow$ to p and **go to** step (2).
2. Generate a unique string id' such that $\forall\langle id, p\rangle \in \mathcal{E}, id \neq id'$, assign $\langle id', p\rangle \cup \mathcal{E}$ to $\mathcal{E}$. □

LEMMA 4.2

For every production memory $\mathcal{P}$ and element class table $\mathcal{E}$ derived from $\mathcal{P}$, there exists a production memory grammar $G_{\mathcal{P}}$ for $\mathcal{P}$.

Proof. By construction, (ALGORITHM 4.3 below). □

ALGORITHM 4.3

Construction of a production memory grammar.

Input. A production memory $\mathcal{P}$ and an element class table $\mathcal{E}$ derived from $\mathcal{P}$.

Output. A production memory grammar $G_{\mathcal{P}} = \langle N_{\mathcal{P}}, \Sigma_{\mathcal{P}}, P_{\mathcal{P}}, S_{\mathcal{P}}\rangle$.

Method. Initially, $G_{\mathcal{P}} = \langle N_{\mathcal{P}}, \Sigma_{\mathcal{P}}, P_{\mathcal{P}}, S_{\mathcal{P}}\rangle = \langle N(\mathcal{P}), \varnothing, \varnothing, S_{\mathcal{P}}\rangle$, where $S_{\mathcal{P}}$ is any non-terminal member of $N_{\mathcal{P}}$. For every production $\langle n, c, a\rangle$ in $\mathcal{P}$ perform steps (1), (2) and (3) below.

1. Initialize the LHS of *temp* to n, the RHS to ε, and **go to** step (2).
2. For all $\langle s_i, p_i^k\rangle \in c$, perform steps (a) and (b) as appropriate, then **go to** step (3).
   (a) If $s_i = +$, then assign id to the ith symbol on the RHS of *temp*, where $\langle id, p_i^k\downarrow\rangle \in \mathcal{E}$, and assign $\{id\} \cup \Sigma_{\mathcal{P}}$ to $\Sigma_{\mathcal{P}}$.
   (b) If $s_i = -$, then assign not_*id* to the ith symbol on the RHS of *temp*, where $\langle id, p_i^k\downarrow\rangle \in \mathcal{E}$, and assign $\{\text{not_}id\} \cup \Sigma_{\mathcal{P}}$ to $\Sigma_{\mathcal{P}}$.
3. Assign $\{temp\} \cup P_{\mathcal{P}}$ to $P_{\mathcal{P}}$. □

THEOREM 4.1

For every Controlled Production System $(\langle \mathcal{P}, W\rangle, G_K)$ there exists a palimpsest parser $\mathcal{F}_P(\langle \mathcal{T}_c, T_0\rangle, d, r, W)$.

Proof. By Construction, (ALGORITHM 4.4, below)

ALGORITHM 4.4

Construction of a palimpsest parser from a controlled production system.

Input. A controlled production system $(\langle \mathcal{P}, W \rangle, G_K)$.

Output. A palimpsest parser $\mathcal{F}_P(\langle \mathcal{T}_c, T_0 \rangle, d, r, W)$.

Method. Perform Steps (1) through (3).

1. Construct $\mathcal{E}$, an element class table derived from $\mathcal{P}$, as in ALGORITHM 4.2.
2. Construct $G_{\mathcal{P}}$, the production memory grammar for $\mathcal{P}$ and $\mathcal{E}$, as in ALGORITHM 4.3.
3. Let $G_{\mathcal{P}K}$ be the palimpsest grammar with a production memory component of $G_{\mathcal{P}}$ and a control component of G_K.
4. Construct (alternate form of) the canonical set of CLR(0) parse tables $\langle \mathcal{T}_c, T_0 \rangle$ for $G_{\mathcal{P}K}$, as in ALGORITHM 3.3.
5. Let d be the palimpsest disambiguation function defined for $G_{\mathcal{P}K}$ and $\mathcal{P}$.
6. Let r be the palimpsest semantics function defined for $G_{\mathcal{P}K}$ and $\mathcal{P}$.
7. Let $\mathcal{F}_P$ be the palimpsest parsing algorithm (ALGORITHM 4.1).
8. Output the resulting palimpsest parser $\mathcal{F}_P(\langle \mathcal{T}_c, T_0 \rangle, d, r, W)$. □

DEFINITION

ALGORITHM 4.4 is called the *palimpsest transformation*. Henceforth, the notation $PT(\mathcal{P}, G_K)$ denotes a palimpsest parser generated from a controlled production system $(\langle \mathcal{P}, W \rangle, G_K)$ by the palimpsest transformation. □

The palimpsest transformation transforms *any* controlled production system into a palimpsest parser. But, the resulting palimpsest parser *may not* interpret the controlled production system correctly if the control grammar is not LR(0); it *may* halt prematurely. Because the solution to this problem is not yet formally defined, it cannot yet be proven that the palimpsest parser generated by the palimpsest transformation is always equivalent to the original controlled production system. However, an equally powerful but less satisfying result can be proven: that for every deterministic Turing machine there exists an equivalent palimpsest parser (THEOREM 4.2).

LEMMA 4.3

A palimpsest parser $PT(\mathcal{P}, G_K)$ applies the action $a(w_{1,k}, W)$ of production $\langle n, c, a\rangle$ only if c is satisfied by W and $w_{1,k} \in W$ is a satisfying list for c.

Proof. A palimpsest parser only applies an action $a(w_{1,k}, W)$ if step (4b) of the palimpsest parsing algorithm applies the equivalent $r(reduce\ i, w_{1,k}, W)$, where $PROD_{\mathcal{P}K}(i) = \langle n, c, a\rangle$. Step (4b) will only be applied if $d(reduce\ i, W)$ has just previously evaluated to $\langle true, w_{1,k}\rangle$ in step (2). By definition, this can occur only if c is satisfied by W and $w_{1,k} \in W^k$ is a satisfying list of c. □

THEOREM 4.2

For every deterministic Turing machine there exists an equivalent palimpsest parser.

Proof. By THEOREM 2.1, for every deterministic Turing machine there must exist an equivalent controlled production system $(\langle \mathcal{P}, W\rangle, G_K)$. Notice, both $\mathcal{P}$ and G_K are the same for all Turing machines, only W, which contains both the tape and the instructions, changes. Therefore, it suffices to show that $PT(\mathcal{P}, G_K)$ correctly interprets $(\langle \mathcal{P}, W\rangle, G_K)$. That is, a legal instantiation is fired iff a legal instantiation exists. LEMMA 4.3 shows that only instantiated productions can be fired. Also, $(\langle \mathcal{P}, W\rangle, G_K)$ is control free, so all productions are always legal. All that remains to be shown is that an instantiation can be found if one exists.

Applying the palimpsest transformation to $(\langle \mathcal{P}, W\rangle, G_K)$ produces an element class table:

$$
\begin{aligned}
\mathcal{E} = \{\ & \langle CONFIG, && w \in config\rangle, \\
& \langle TAPE, && w \in tape\rangle, \\
& \langle LEFT_INSTR, && w \in instr \wedge move(w) = left\rangle, \\
& \langle RIGHT_INSTR, && w \in instr \wedge move(w) = right\rangle, \\
& \langle STAY_INSR, && w \in instr \wedge move(w) = left\rangle, \\
& \langle INSTR, && w \in instr\rangle\ \}
\end{aligned}
$$

and a palimpsest grammar $G_{\mathcal{P}K}$, below, with CLR(0) parse tables in FIGURE 4.4.

(1) LOOP → LENGTHEN_TAPE LOOP

(2) LOOP → DO_LEFT LOOP

(3) LOOP → DO_RIGHT LOOP
(4) LOOP → DO_STAY LOOP
(5) LOOP → HALT_CONDITION
(6) LENGTHEN_TAPE → config not_tape
(7) DO_LEFT → config tape left_instr
(8) DO_RIGHT → config tape right_instr
(9) DO_STAY → config tape stay_instr
(10) HALT_CONDITION → config tape not_instr

By inspection of FIGURE 4.4, after firing any production other than ***halt_condition***, the parser will be in a configuration with the topmost table on the parse stack (i.e., T_{top}) equal to one of $T_0, T_2, T_3, T_4,$ or T_5. From all of these states, reductions by rules (6) through (10), corresponding to the productions ***do_left, do_right, do_stay,*** and ***lengthen_tape,*** are reachable by applying parse actions in the table as follows:

Rule (6): *shift* ***config,*** *shift* ***not_tape,*** *reduce 6.*

Rule(7): *shift* ***config,*** *shift* ***not_tape,*** backtrack and *shift* ***tape,*** *shift* ***left_instr,*** *reduce 7.*

Rule(8): *shift* ***config,*** *shift* ***not_tape,*** backtrack and *shift* ***tape,*** *shift* ***left_instr,*** backtrack and *shift* ***right_instr,*** *reduce 8.*

Rule(9): *shift* ***config,*** *shift* ***not_tape,*** backtrack and *shift* ***tape,*** *shift* ***left_instr,*** backtrack and *shift* ***right_instr,*** backtrack and *shift* ***stay_instr,*** *reduce 9.*

Rule(10): *shift* ***config,*** *shift* ***not_tape,*** backtrack and *shift* ***tape,*** *shift* ***left_instr,*** backtrack and *shift* ***right_instr,*** backtrack and *shift* ***stay_instr,*** backtrack and *shift* ***not_instr,*** *reduce 10.*

After applying any of rules (6) through (9), the parser returns to one of the initial configurations with T_{top} equal to one of $T_0, T_2, T_3, T_4,$ or T_5. After applying rule (10) (i.e., ***halt_condition***), no other productions have satisfied conditions, and the palimpsest parser begins reducing the parser stack and eventually applies the *accept* action. If an instantiation exists, then it will be found unless another instantiation also exists and is fired in its place. Either way, an instantiation is fired iff an instantiation exists. □

	1	2	3	4	L	LT	DL	DR	DS	HC	c	t	l	r	s	n	m
T_0	Sc	–	–	–	T_1	T_2	T_3	T_4	T_5	T_6	T_7	–	–	–	–	–	–
T_1	A	–	–	–	A	–	–	–	–	–	–	–	–	–	–	–	–
T_2	Sc	–	–	–	T_8	T_2	T_3	T_4	T_5	T_6	T_7	–	–	–	–	–	–
T_3	Sc	–	–	–	T_9	T_2	T_3	T_4	T_5	T_6	T_7	–	–	–	–	–	–
T_4	Sc	–	–	–	T_{10}	T_2	T_3	T_4	T_5	T_6	T_7	–	–	–	–	–	–
T_5	Sc	–	–	–	T_{11}	T_2	T_3	T_4	T_5	T_6	T_7	–	–	–	–	–	–
T_6	R5	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_7	Sn	St	–	–	–	–	–	–	–	–	–	T_{13}	–	–	–	T_{12}	–
T_8	R1	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_9	R2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{10}	R3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{11}	R4	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{12}	R6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{13}	Sl	Sr	Ss	Sm	–	–	–	–	–	–	–	–	T_{14}	T_{15}	T_{16}	–	T_{17}
T_{14}	R7	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{15}	R8	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{16}	R9	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
T_{17}	R10	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

FIGURE 4.4. CLR(0) Parse Tables for Turing Machines[128]

[128] where, L ≡ LOOP, LT ≡ LENGTHEN_TAPE, DL ≡ DO_LEFT, DR ≡ DO_RIGHT, DS ≡ DO_STAY, HC ≡ HALT_CONDITION, c ≡ config, t ≡ tape, l ≡ left_instr, r ≡ right_instr, s ≡ stay_instr, n ≡ not_tape, m ≡ not_instr

5. Analysis of Palimpsest Parser Performance

In this chapter, the time and space efficiency of palimpsest parsers is examined. As is common in the literature[129,130], production system performance is based upon two measurements: the time required to fire a single instantiation, and run-time storage requirements. The focus is on two quantifiable production system characteristics that appear to have the greatest effect upon performance: the size of production memory, and the size of working memory. Various implementation issues are discussed in order to provide a basis for later performance claims. An overview of the processing required to fire an instantiation is presented, followed by the best, worst, and expected case time costs and the expected space costs with respect to production memory and working memory sizes. Empirical results for simulated controlled production systems are presented that support these time and space cost calculations. The chapter concludes with a discussion of potential optimizations.

5.1. Implementation Issues

Heretofore, palimpsest parsers have been described abstractly to emphasize concepts and ideas rather than any specific implementation. However, one possible implementation of the more abstract palimpsest parser data structures and processes is presented to justify performance claims made later in this chapter .

---

129 Forgy, 1979.

130 McDermott, Newell, and Moore, 1978.

5.1.1. Implementing Working Memory

As defined in CHAPTER 4, working memory in a palimpsest parser is represented as a union of sets, each of which contains the subset of an element class found in that working memory. One additional set contain those working memory elements that do not belong to any element class, and so do not contribute to the interpretation. Because a working memory element may belong to many element classes, implementation of such a working memory representation is not necessarily obvious. This section presents one possible implementation.

5.1.1.1. A Working Memory Data Structure

One possible working memory data structure for palimpsest parsers is shown in FIGURE 5.1. The element classes defined by a production memory index an array of linked lists. Each such list contains exactly one member record for every working memory element in working memory that is a member of that element class. Every member record contains three pointers:

1. A pointer to a member record in another element class list that points to the same working memory element. The lists defined by these pointers are circular,
2. A pointer to the actual working memory element, and
3. A pointer to the next member record in the list.

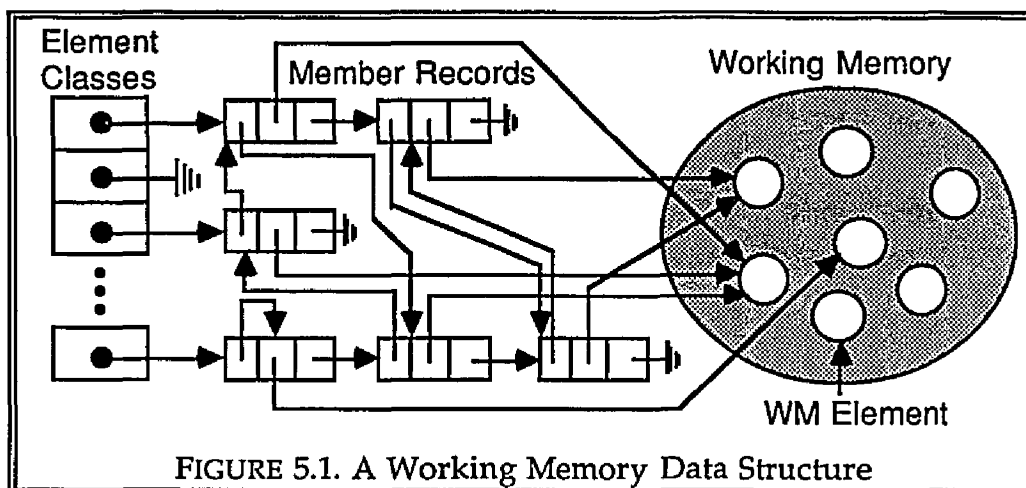


FIGURE 5.1. A Working Memory Data Structure

5.1.1.2. Adding Elements to Working Memory

Adding a new element to the working memory data structure of FIGURE 5.1 is a three step process:

1. Create the working memory element. The actual location of a working memory element (i.e., on the heap, in an array, etc.) is irrelevant, as long as a pointer (or index) to it is known.
2. Classify the working memory element. That is, determine to which element classes it belongs.
3. For every element class to which the element belongs, add a new member record to the beginning of the list indexed by that element class. Also, all member records added during this step must be linked into a circular list.

Ideally, steps (2) and (3) are interleaved. Whenever it is determined that the working memory element belongs to an element class, a member record is added to the list indexed by that element class.

Step (2) can be implemented in a brute force manner by evaluating all element class specifications on a given working memory element. However, such a simple implementation typically performs many redundant and useless tests. For most production systems, the time required to classify working memory elements can be significantly reduced, as follows:

- Use nesting to avoid redundant evaluation of predicates.
- Use selection statements that support mutual exclusion (e.g., *if-then-else* statements).
- Use multi-branch selection statements (e.g., *case* statements in Pascal and Ada) to test equality predicates.
- Re-arrange the order of predicate testing to support the above.

Each of the above optimizations is discussed below.

Often, one element class specification will differ from another by only one of many predicates. Redundant evaluation of the shared predicates can be avoided by nesting predicate tests. For example, the four predicate tests in the following Ada fragment

```
if in_bay(w) = none and size(w) ≥ 66 then
   add w to the list for element class X;
end if;
if in_bay(w) = none and size(w) > 33 then
   add w to the list for element class Y;
end if;
```

could be reduced to the equivalent three predicate tests in

```
if in_bay(w) = none then
   if size(w) ≥ 66 then
      add w to the list for element class X;
   end if;
   if size(w) > 33 then
      add w to the list for element class Y;
   end if;
end if;
```

Nesting predicate tests in this way significantly reduces the number of predicate tests performed, especially when shared predicates are tested before distinguishing predicates.

Many predicate tests are mutually exclusive, for example ***age≥40, age≤20,*** and ***age=30***. Should any of these predicates evaluate true for a working memory element, the others must evaluate to false; further testing of such predicates is wasteful. This mutual exclusion may be exploited by the use of selection statements such as the Ada ***if-then-else*** statement. For example, the following sequence of ***if*** statements

```
if age(w) >= 40 then
   add w to the list for element class A;
end if;
if age(w) <= 20 then
   add w to the list for element class B;
end if;
          • • •
if age(w) = 30 then
   add w to the list for element class Z;
end if;
```

may be replaced by an equivalent series of ***if-then-else*** statements as follows:

```
if age(w) >= 40 then
    add w to the list for element class A;
else if age(w) <= 20 then
    add w to the list for element class B;
        • • •
else if age(w) = 30 then
    add w to the list for element class Z;
end if;
```

Such "short circuiting" of predicate testing significantly reduces the number of predicate tests performed, especially on those tests that occur early in the sequence. Moreover, ordering the predicate tests according to their likelihood of evaluating to *true* also improves performance.

Many predicates are mutually exclusive because they all test to see if the same attribute value is equal to some constant. For example, the predicates *type=dog, type=cat,* and *type=mouse* are mutually exclusive because the *type* attribute can have exactly one value. When these constant values can be enumerated the predicates are amenable to a particularly efficient testing construct: the multi-branch selection statement (e.g., *case* statements in Pascal and Ada), that can test any number of such mutually exclusive predicates in constant time. For example, the following *case* statement tests the *type* predicates above.

```
case type(w) of
    when dog     ⇒ add w to the list for element class A;
    when cat     ⇒ add w to the list for element class B;
                     • • •
    when mouse   ⇒ add w to the list for element class Z;
end case;
```

The use of multi-branch selection statements significantly reduces the cost of predicate testing, especially when the more expensive predicate tests are nested within multi-branch selections. Because a category must occur in every pattern, it should always be tested by the outermost multi-branch selection.

The effectiveness of the above methods depends upon the order in which predicates within a pattern are tested. This suggests that a judicious re-ordering of these tests to allow the application of the above methods would signifi-

cantly increase that effectiveness. Predicate tests within a pattern should be ordered, from first to last, as follows:

1. Predicates amenable to multi-branch selection should be evaluated first; specifically, predicates that test for the equality of attribute values and enumerable constants.
2. Predicates in one pattern that have mutually exclusive counterparts in other patterns should be evaluated before predicates that do not. These mutually exclusive predicates should be ordered the same in all such patterns to facilitate the use of selection statements.
3. Remaining predicates should then be ordered by frequency; predicates that occur in many patterns should be evaluated before predicates that are unique to a pattern. This facilitates nesting to reduce redundant evaluation.

At present, the ordering of predicates is the responsibility of the production system programmer.

5.1.1.3. Removing Elements from Working Memory

Removing a working memory element from the working memory data structure in FIGURE 5.1 requires two steps:

1. Remove from all element class lists all member records that point to the working memory element.
2. Remove the actual working memory element from working memory.

Although a working memory element may belong to many element classes, there is only one of every element. Removal of a working memory element requires that all member records that reference it must also be removed. Because all member records that point to the same working memory element are linked in a circular list, steps (1) and (2) are straightforward. For example, when the working memory element is deleted, all member records on the circular list can be marked for later removal when the element class lists are traversed.

5.1.2. Implementing Disambiguation Functions

A palimpsest disambiguation function d can be implemented as a function with case statements indexed by parsing actions as in FIGURE 5.2.

The code associated with a shift action, say *shift x,* assigns *true* to d if the element class list indexed by x is non-empty, and assigns *false* to d otherwise. When a reduce action, such as *reduce i,* is passed to d the result depends upon the nature of the grammar rule denoted by i. If i denotes a rule in the control component of the palimpsest grammar, then d is automatically assigned *true.* If i denotes a rule in the production component of the palimpsest grammar, such as

$$\text{LOADITEM} \rightarrow \text{anyitem} \;\; \text{not_openbay} \;\; \text{closedbay}$$

then d must evaluate an expression, such as

$$\exists\langle w_1, w_3\rangle \in (anyitem \cap W) \times (closedbay \cap W) \ni [space(w_3) \geq size(w_1) \\ \wedge \forall w_2 \in openbay \cap W, \neg(space(w_2) \geq size(w_1))].$$

to determine whether or not the many-element component of the production is satisfied by working memory.

```
function d (action:in shift_or_reduce; which:in ec_or_rule) returns boolean is
   — This is the outer skeleton of the disambiguation function d.
begin
   return true;
   case action is
      when  shift  ⇒
         if element_class_list(which).first = null then
               return false;
      when  reduce ⇒
         if is_production_memory_rule(which) then
               — The evaluation of many-element predicates is performed
               — here by the code found in FIGURE 5.3.
         end if;
   end case;
end;  {function d}
```

FIGURE 5.2. An Ada Implementation of Function d

```
-- The following is inserted within the case statement of FIGURE 5.2. --

test_sign := positive;
satisfied := true;
init_list(sstack, stk_top, k, positive, no_more);

loop
  case which is
                          .
                          .
                          .
      -- The following code tests the many element predicates of
      -- the production corresponding to rule number i in the
      -- palimpsest grammar. Similar case alternatives exist for
      -- all productions.
      when i =>
         if test_sign = positive then
           satisfied := closedbay.space >= anyitem.size;
         else
          satisfied := not ((not_openbay = nil)
                        and (not_openbay.space >=
                              anyitem.member.size));
         end if;
                          .
                          .
                          .
  end case;
  if (test_sign = positive) then
     if not satisfied then
        next_positive_list(sstack, stk_top, k, no_more);
     else
        test_sign := negative;
        init_list(sstack, stk_top, k, positive, no_more);
     end if;
  else
     if not satisfied then
        next_negative_list(sstack, stk_top, k, no_more);
     else
        test_sign := positive;
        next_positive_list(sstack, stk_top, k, no_more);
     end if;
  end if;
  exit when no_more;
end loop;
return satisfied;
```

FIGURE 5.3. Testing Many-Element Predicates in Function *d*[131]

131 The element class names used as records (e.g., ***anyitem, not_openbay,*** and ***closedbay***) within the context of a grammar rule are automatically converted into references to records on the semantics stack ***sstack*** prior to compilation of the palimpsest parser. For example, the occurrences of ***closedbay*** for production *i* will become ***sstack(top).member;*** similarly, occurrences of ***not_openbay*** will become ***sstack(top-1).member;*** and occurrences of ***openbay*** will become ***sstack(top-2).member.***

```
procedure next_positive_list   (sstack   : in out semantics_stack_type;
                                rhs_size : in pattern_count_range;
                                top      : in sstack_range;
                                all_empty: in out boolean) is
   ix : stack_range;
begin
   all_empty := false;
   ix        := top - rhs_size + 1;
   while ((sstack(ix).member.next = null) or (sstack(ix).sign = negative))
     and (ix <= top) loop
   begin
      sstack(ix).member := elt_class_list(sstack(ix).elt_class).first;
      ix := ix + 1;
   end loop;
   if ix > top then
      all_empty := true;
   else
      sstack(ix).member := sstack(ix).member.next;
   end if;
end next_positive_list;

procedure next_negative_list   (sstack   : in out semantics_stack_type;
                                rhs_size : in pattern_count_range;
                                top      : in sstack_range;
                                all_empty: in out boolean) is
   ix : stack_range;
begin
   all_empty := true;
   for ix in (top - rhs_size + 1) .. top loop
      if (sstack(ix).member.next <> null) and (sstack(ix).sign = negative) then
         sstack(ix).member := sstack(ix).member.next;
         all_empty := false;
      end if;
   end loop;
end next_negative_list;

procedure init_list   (sstack   : in out semantics_stack_type;
                       rhs_size : in pattern_count_range;
                       top      : in sstack_range;
                       sign     : in positive_or_negative;
                       all_empty: in out boolean) is
   ix : stack_range;
begin
   all_empty := true;
   for ix in (top - rhs_size + 1) .. top loop
      if sstack(ix).sign = sign then
         sstack(ix).member := elt_class_list(sstack(ix).elt_class).first;
      end if;
      if sstack(ix).member /= null then
         all_empty := false;
      end if;
   end loop;
end init_list;
```

FIGURE 5.4. Procedures Called by Function $\mathcal{A}$

One way to implement such an expression is shown in FIGURE 5.3. It is assumed that the top k records on a semantics stack ***sstack*** are associated with the k patterns on the RHS of the production. Each of these records contains: an element class name or index ***elt_class*** specified by the pattern, a boolean ***sign*** denoting whether the pattern is negated, and an access variable (i.e., a pointer) ***next*** (initially ***null***) to the next member record in the appropriate element class list. A procedure called ***next_positive_list*** modifies the topmost k ***next*** pointers on ***sstack*** such that successive calls will cycle through all possible lists of positive working memory elements in the element class lists specified by the topmost k ***elt_class*** and ***sign*** values on ***sstack***. A similar procedure called ***next_negative_list*** cycles through all possible lists of negative working memory elements.

Again, a multi-branch selection or **case** statement is used to select the proper code for $\mathcal{d}$. However, much of the code for each case is independent of the expression being evaluated, and has been moved outside of the **case** statement. Remember from CHAPTER 4 that $\mathcal{d}$ returns a list in addition to a boolean value. In the implementation above, the k elements of this list are found, one each, in the topmost k records on ***sstack*** by following the ***next*** access link to a member record, and then the working memory element.

5.2. Palimpsest Parser Time Costs

5.2.1. Overview of Palimpsest Parser Execution

Production system interpretation by palimpsest parser, as depicted in FIGURE 5.5, involves the repeated application of three distinct processes:

1. Find an instantiation
2. Fire an instantiation
3. Apply control rules

each of which, in turn, may be composed of various other sub-processes. The total time cost for palimpsest parsers with respect to a production system characteristic is the sum of the time costs for these three processes.

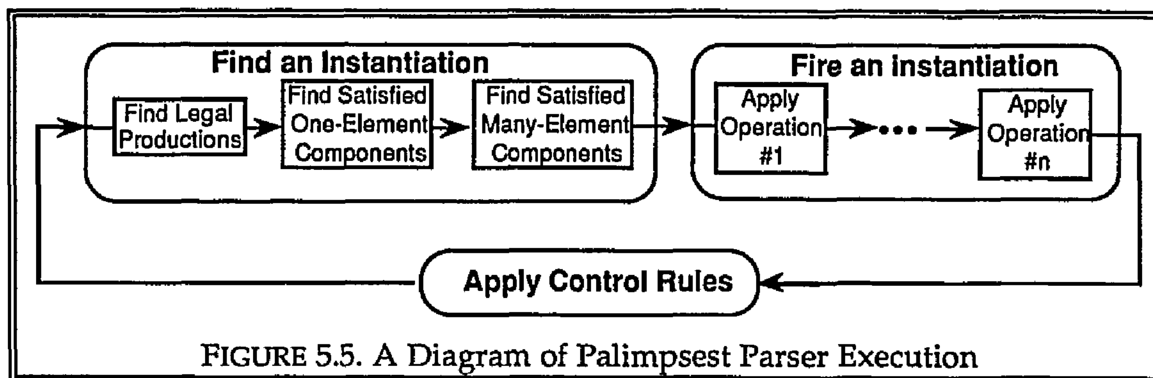


FIGURE 5.5. A Diagram of Palimpsest Parser Execution

5.2.1.1. Finding an Instantiation

The process of finding an instantiation by a palimpsest parser may be logically divided into three phases:

1. Find legal productions
2. Determine whether the one-element components are satisfied
3. Determine whether the many-element components are satisfied

These three phases are repeated until an instantiation is found or all legal productions have been examined. The time cost of finding an instantiation is then the sum of the time costs for these three phases.

A palimpsest parser is constrained by its control grammar to search for instantiations of legal productions only. Illegal productions are never examined. This constraint is automatic and requires no additional processing.

The one element component of a pattern is satisfied if the name of the element class it specifies has been shifted onto the parse stack by the palimpsest parser. Such a shift action can occur only if the intersection of working memory and the element class is non-empty. If symbols corresponding to all patterns in a condition are on the top of the stack, then the one-element component of that condition must be satisfied. If however, one such intersection is empty, then a palimpsest parser will backtrack to a previous state and try a different parsing action. For a palimpsest parser the time cost of evaluating one-element predicates is independent of production memory and working memory sizes; and the time costs of shift and backtrack operations are con-

stant; so the time cost of this phase is proportional to the numbers of shift and backtrack operations applied.

A condition is satisfied if working memory contains a list of elements that satisfies all of the positive patterns in the condition, but does not satisfy any of the negative patterns in the condition. As described in SECTION 5.1.2, a palimpsest disambiguation function performs this phase by evaluating the many-element components of *k*-predicates on members of specific element classes. The time cost of evaluating many-element predicates is independent of production memory and working memory sizes; so the time cost of this phase is proportional to the number of many-element predicate evaluations performed.

5.2.1.2. Firing an Instantiation

Production actions in palimpsest parsers are composed of sequences of *ADD* and *REM* operations. A working memory element is added to working memory by evaluating all element class specifications on that element and adding it to the appropriate element class lists. Similarly, a working memory element is removed from working memory by evaluating all element class specifications on that element and removing it from the appropriate element class lists. The time cost of firing an instantiation by a palimpsest parser is then the product of the number of operations in the action and the time costs of those operations.

5.2.1.3. Applying Control Rules

As described above in SECTION 5.2.1.1, the use of a control grammar to constrain the search for satisfied productions is automatic and incurs no additional time cost. However, after an instantiation has fired, the application of zero or more control rules may be necessary to place the palimpsest parser into a new state in which productions may be fired. The time costs of such control rule applications are independent of the size of production memory. The time cost of applying control rules after firing an instantiation is then proportional to the number of rules applied.

5.2.2. Effects of Production Memory Size on Time

Of all quantifiable production system characteristics, the effect of production memory size (P) on execution times most markedly differentiates palimpsest parsers and other production system architectures. Although the best and worst case time costs with respect to **P** are equivalent for palimpsest parsers and other production system architectures, the expected time complexity of palimpsest parsers with respect to **P** is $O(1)$ while other architectures are at least $O(\mathbf{P})$.

5.2.2.1. Worst Case Effect of Production Memory Size on Time

The maximum number of shift operations that can be applied in the process of finding an instantiation is bounded by $\sum_{i=1}^{P} C_i$, where C_i is the number of patterns in the condition of production *i*. Since something must be shifted before it can be backtracked over, the maximum number of backtrack operations is also bounded by $\sum_{i=1}^{P} C_i$.

The maximum number of positive, many-element predicates evaluated in the process of finding an instantiation is bounded by $\sum_{i=1}^{P} W^{C_i^+}$, where C_i^+ is the number of positive patterns in production *i*. That is, one test is made for every production and every possible list of working memory elements of size C_i^+. The maximum number of negative, many-element predicates evaluated is $\sum_{i=1}^{P} (W^{C_i^+} \bullet W \bullet C_i^-)$, where $C_i^- = C_i - C_i^+$ is the number of negative patterns in production *i*[132]. That is, for every list of working memory elements found to satisfy positive patterns, it must be verified that that list satisfies none of

[132] Notice that the summation expression for negative many-element predicate evaluations contains $(W^{C_i^+} \bullet W \bullet C_i^-)$ instead of $(W^{C_i^+} \bullet W^{C_i^-})$. Because negative patterns are never referenced by other patterns, working memory elements can be tested by one negative pattern independently of the tests of other negative patterns; combinations are irrelevant (cf. FIGURE 5.3).

the C_i^- negative patterns. A condition containing C_i patterns requires the most computation when all of its patterns are positive, that is $\sum_{i=1}^{P} W^{C_i}$, where $C_i = C_i^+$.

The maximum number of one-element predicates that can be evaluated by a classification procedure for every element added to or removed from working memory is bounded by $\sum_{i=1}^{P} C_i$. That is, all pattern predicates in production memory specify different but not mutually exclusive element classes, and none of the optimizations of SECTION 5.1.1.2 are applicable.

The maximum number of control rules that may be applied cannot be specified *a priori*. However, for controlled production systems, in which every control rule represents a plan for achieving some task, the average control rule RHS should require many (at least two) productions to fire. A generous upper bound on the number of control rules applied after firing a production is then $\log_2 P$.

The worst case time cost of finding and firing a single instantiation by a palimpsest parser is then the sum of the above maxima multiplied by the appropriate constants; specifically

$$k_1 \bullet \sum_{i=1}^{P} C_i + k_2 \bullet \sum_{i=1}^{P} C_i + k_3 \bullet \sum_{i=1}^{P} W^{C_i} + k_4 \bullet k_5 \bullet (\sum_{i=1}^{P} C_i + k_6) + \log_2 P \bullet k_7$$

where:

- k_1 is the time cost of performing a shift operation.
- k_2 is the time cost of performing a backtrack operation.
- k_3 is the time cost of evaluating a many-element predicate.
- k_4 is the number of operations in the applied action.
- k_5 is the time cost of evaluating an element-class specification.
- k_6 is the time cost of adding or removing a working memory element.
- k_7 is the time cost of applying control rules.

If C_{max} is the maximum number of patterns in any condition in production memory, then the above expression is bounded by

$$k_1 \bullet P \bullet C_{max} + k_2 \bullet P \bullet C_{max} + k_3 \bullet P \bullet W^{C_{max}} + k_4 \bullet k_5 \bullet (P \bullet C_{max} + k_6) + \log_2 P \bullet k_7$$

which can be simplified to

$$P \bullet [(k_1 + k_2 + k_4 \bullet k_5) \bullet C_{max} + k_3 \bullet W^{C_{max}}] + k_4 \bullet k_5 \bullet k_6 + \log_2 P \bullet k_7.$$

The resulting worst case time complexity with respect to **P** is $O(\mathbf{P})$.

The kind of production system necessary for a palimpsest parser to exhibit this worst case behavior is unrealistic, especially for large production systems. Many of the required production system characteristics are even mutually exclusive. For example, in order for the **P** to appear in most terms, all productions in production memory must be examined before any instantiation is found. This requires that:

- The production system be control free.
- All conditions have satisfied one-element components.
- For all productions except the last one tested, the many-element component of the condition is unsatisfiable by working memory.
- For the last production tested, the many-element component of the condition is satisfied only by the last list of working memory elements tested.
- Every pattern specifies a different element class, but no predicates in element class specifications can be mutually exclusive or amenable to other optimizations.

However, a very basic assumption is that no realistic, large production system will be control free. Also, in order for $W^{C_{max}}$ many-element predicate evaluations to be performed for every production, every element in working memory must always belong to every element class; but, this contradicts the above requirement that every pattern specifies a different element class.

In order for the time costs of *ADD* and *REM* operations to be proportional to $\mathbf{P} \bullet \mathbf{C}$, then again, every pattern must specify a different element class, but no predicates in element class specifications can be mutually exclusive or

amenable to other optimizations. However, this contradicts a basic assumption, that the rate of growth of new element class specifications is sublinear with respect to production memory size[133].

In order for $\log_2 P$ control rules to be applied after every instantiation firing, the production system must specify a maximum amount of control. However, this contradicts the above assumption that the production system is control free.

5.2.2.2. Best Case Effect of Production Memory Size on Time

The minimum number of shift operations performed by a palimpsest parser in the process of finding an instantiation is C_i, the number of patterns in the first condition examined. In the best case no backtracking is required, so the minimum number of backtrack operations performed by a palimpsest parser is zero.

The minimum number of many-element predicate evaluations made by a palimpsest parser in the process of finding an instantiation is one[134]. That is, for the first production examined, the disambiguation function $\mathcal{d}$ returns true for the first list of working memory elements to be tested.

The minimum number of one-element predicates evaluated by a classification procedure for every element added to or removed from working memory is bounded by a constant. That is, if the pattern predicates in production memory specify element classes such that the optimizations of SECTION 5.1.1.2 are applicable, the cost of classifying a working memory element is independent of the size of production memory.

The minimum number of control rules that can be applied after firing an instantiation is zero.

---

133 cf. SECTION 5.2.2.3.

134 This assumes that the condition has only positive patterns (otherwise the number of evaluations would be 2), and that the condition has a non-vacuous many-element component (otherwise, the number of evaluations would be 0).

The best case time cost of finding and firing a single instantiation by a palimpsest parser is then the sum of the above minimums multiplied by the appropriate constants; specifically

$$k_1 \bullet 1 + k_2 \bullet 0 + k_3 \bullet 1 + k_4 \bullet k_5 \bullet (1 + k_6) + k_7 \bullet 0$$

where the minimum size of any production condition is one, and k_1 through k_7 are the constants defined in SECTION 5.2.2.1. This expression may be simplified to:

$$k_1 + k_3 + k_4 \bullet k_5 \bullet (1 + k_6)$$

The resulting best case time complexity with respect to **P** is $O(1)$.

Again, it is very unlikely that any real palimpsest parser will exhibit this minimal time cost. However, the best case requirements are much less stringent than those for the worst case; specifically:

- The first production examined is instantiated by the first list of working memory elements evaluated.
- The time required to classify a working memory element is independent of the size of production memory.

No assumptions about the size of the production system, the amount of procedural control or the form of productions other than the one fired are required.

5.2.2.3. Expected Effect of Production Memory Size on Time

Before the expected effect of production memory size on time costs can be predicted, it is first necessary to make some assumptions about what constitutes a typical production system. The following paragraphs outline and justify such assumptions and assess their effect on the overall time costs. The expected effect of production memory size on time is then discussed.

The average number of legal productions over all states in a palimpsest parser is constant with respect to the number of productions. That is, the amount of control information grows in proportion to the size of production memory. New productions are added to a production system to either refine

its existing capabilities or to provide new ones. Productions that are added to provide new capabilities require new control information to specify their scope within the existing system. Productions that are added to refine existing capabilities eventually generate the need for additional control to be imposed because humans naturally divide large, complex problems into manageable subproblems. The size of these subproblems tends to be independent of the size of the overall problem and is primarily related to the complexity that an individual can deal with at any one time[135]. Each subproblem is then represented by a set of legal productions. Because only legal productions are examined by a palimpsest parser, the time required to find an instantiation is proportional to the number of legal productions, and is independent of the size of production memory.

The number of unique, one-element predicates evaluated by a classification procedure grows sublinearly with respect to the size of production memory. That is, the likelihood of a new production specifying a new one-element predicate decreases as the number of predicates in use increases[136]. Consider adding new productions to a large, existing production system. Most likely, the production is being added to correct a deficiency, or to add a new capability. In both cases, the production must examine and modify pre-existing working memory elements in order to mesh with the existing system; and the larger the original production system, the more likely that the one element components of the new patterns have occurred in other productions. In addition, the likelihood that the optimizations of SECTION 5.1.1.2 apply to any new one-element predicate increases as the number of predicates in use increases. For large production systems, the number of one-element predicates evaluated in order to add or remove a working memory element should tend to remain constant.

Iterations defined by the control component of a palimpsest grammar must reference productions; that is, instantiations must be fired on each cycle of the iteration. The purpose of the control grammar is to control the order of instantiation firing, not to perform extraneous processing. The only

135 Forgy, Gupta, Newell, and Wedig, 1984, p. 118.
136 Forgy, 1979, p. 105.

justifiable purpose for iteration in a control grammar is to repeatedly fire some sequence of instantiations. The number of control rules applied after firing an instantiation should then be bounded by some constant.

Taking into account the above assumptions, the expected time cost of finding and firing an instantiation by a palimpsest parser is bounded by the following constant expression

$$k_1 \bullet k_8 \bullet C_{max} + k_2 \bullet k_8 \bullet C_{max} + k_3 \bullet k_8 \bullet W^{C_{max}} + k_4 \bullet k_5 \bullet (k_9 + k_6) + k_7$$

where

k_8 is the average number of legal productions
k_9 is the maximum number of one-element predicates evaluated by the classification procedure.

The expected time complexity of palimpsest parsers with respect to **P** is $O(1)$. The expected time complexities of most other production system architectures with respect to **P** are at least $O(\mathbf{P})$[137].

5.2.3. Effects of Working Memory Size on Time

This section describes the best, worst, and expected case time costs of finding and firing an instantiation with respect to the size of working memory.

5.2.3.1. Worst Case Effect of Working Memory Size on Time

The expression for the worst case time cost of palimpsest parsers presented in SECTION 5.2.2.1 contains only one term that depends upon working memory size; that is, $k_3 \bullet W^{C_{max}}$. The worst case time complexity of palimpsest parsers with respect to working memory size **W** is then $O(\mathbf{W}^{C_{max}})$.

137 Forgy, 1982.

5.2.3.2. Best Case Effect of Working Memory Size on Time

The expression for the best case time cost of palimpsest parsers presented in SECTION 5.2.2.2 contains no terms that depend upon working memory size. The best case time complexity of palimpsest parsers with respect to working memory size **W** is then $O(1)$.

5.2.3.3. Expected Effect of Working Memory Size on Time

The expected time complexity of palimpsest parsers with respect to working memory size **W** is difficult to determine. A large working memory can, in the worst case, greatly increase the execution time of a palimpsest parser. However, a large working memory also increases the likelihood that positive patterns, the patterns that contribute the $W^{C_{max}}$ term to the worst case time complexity, will be satisfied. Empirical tests have shown that execution times tend to be independent of **W**[138]. The expected time complexity of palimpsest parsers with respect to the size of working memory **W** is then $O(1)$.

5.3. Palimpsest Parser Space Costs

5.3.1. Overview of Palimpsest Parser Composition

The space cost for a palimpsest parser is the sum of the static space costs (i.e., costs dependent upon **P**) for the following palimpsest parser components:

- The compiled palimpsest parsing algorithm
- The compressed CLR(0) parse tables
- The compiled element class classification procedure
- The compiled disambiguation function d
- The compiled semantics function r

---

138 cf. SECTION 5.4.

and the dynamic space costs of the runtime working memory data structure (i.e., costs dependent upon **W**). All of these individual space costs are dependent upon the palimpsest parser implementation chosen; that of SECTION 5.1 is assumed.

5.3.1. Effects of Production Memory Size on Space

The palimpsest parsing algorithm is independent of the palimpsest grammar that it interprets. Hence, the space cost incurred by the palimpsest parsing algorithm is a constant.

The space cost of CLR(0) parse tables is difficult to determine, *a priori*. First, these tables tend to be very sparse, and are amenable to a number of space reducing transformations[139]. Based on empirical tests of simulated controlled production systems, it appears that space grows linearly with respect to **P**.

As discussed in SECTION 5.2.2.3, the number of unique element classes and thus the size of the compiled element class classification procedure should grow sub-linearly with respect to **P**. A generous estimate would be a size proportional to $\log_2 \mathbf{P}$.

Since exactly one case alternative is added to *d* and *r* for every production, space cost of these functions is proportional to **P**. Combining all of the above costs, the expected static space cost of a palimpsest parser is:

$$k_1 + k_2 \bullet \mathbf{P} + k_3 \bullet \log_2 \mathbf{P} + k_4 \bullet \mathbf{P} + k_5 \bullet \mathbf{P}$$

which reduces to:

$$\mathbf{P} \bullet (k_2 + k_4 + k_5) + k_3 \bullet \log_2 \mathbf{P} + k_1$$

where:

- k_1 is the space cost of the palimpsest parsing algorithm.
- k_2 is the average space cost for each production's contribution to the CLR(0) parse tables.

139 Dencker, Dürre, and Heuft, 1984.

k_3 is the space cost of the element class classification procedure.
k_4 is the space cost of one **case** alternative in a disambiguation function *d*.
k_5 is the space cost of one **case** alternative in a semantics function *r*.

The space complexity of palimpsest parsers with respect to **P** is *O*(**P**).

5.3.2. Effects of Working Memory Size on Space

The number of elements in working memory has no effect on the static space costs above, but only on the space cost of the implementation dependent working memory data structure itself. Assuming the data structure of SECTION 5.1.1.1, the total dynamic space cost is a function of the number of member records and the number of the working memory elements. Every working memory element appears in memory only once, so the space cost incurred by the working memory elements is proportional to **W**. Similarly, with respect to **W**, the space cost incurred by member records is proportional to $\mathbf{W} \bullet k_7$, where k_7 is the average number of element classes to which each working memory element belongs. The total dynamic space cost of a palimpsest parser is:

$$\mathbf{W} \bullet (k_6 + k_7)$$

where:

k_6 is the average space cost of a working memory element.
k_7 is the average number of element classes containing each working memory element.

The space complexity of palimpsest parsers with respect to **W** is *O*(**W**).

5.4. Empirical Tests of Palimpsest Parsers

This section presents the results of empirical tests to support the theoretical time and space cost calculations above. These results were obtained by testing automatically generated controlled production systems. Note, the

methodology used to construct such controlled production systems makes a number of assumptions about typical controlled production systems characteristics, and the validity of those assumptions have not been proven[140].

5.4.1. Description of The Testing Methodology

Ideally, a large number of real world controlled production systems would be analyzed to determine the effects of production memory size and working memory size on time and space costs. Unfortunately, finding non-proprietary production systems, of any kind, for use in such a study is very difficult. And, even if such an array of production systems were available, distinguishing the effects of individual production system characteristics on the total time and space costs would also be difficult. As an alternative, the BUILDER program creates controlled production systems for analysis.

The BUILDER program takes as input a number of quantifiable controlled production system characteristics and produces a syntactically correct, but semantically meaningless controlled production system[141] that displays those characteristics. Such characteristics include:

- The number of productions (P).
- The number of working memory elements (W).
- The mean number of patterns per condition (C).[142]
- The mean number of control rules with the same LHS (G). This estimates module size.
- The ratio of the number of element classes to the number of patterns.

---

140 These assumptions include: the distribution of the number of patterns per condition; the distribution of the number of operations per action; the distribution of working memory elements among element classes, etc.

141 Actually, instead of creating a controlled production system that must then be transformed into a palimpsest parser, the BUILDER program creates a palimpsest grammar, disambiguation function, and reduce semantics function directly.

142 The standard deviation of all mean values is also be specified to define the width of a normal frequency distribution.

- The mean number of element classes to which each working memory element belongs.
- The ratio of positive to negative patterns in conditions.
- The probability that any many-element component in a positive pattern will evaluate to *true*.
- The probability that any many-element component in a negative pattern will evaluate to *true*.

A number of other characteristics are available; however, for the purposes of this chapter, attention will be restricted to the effects of the most significant characteristics: **P**, **G**, and **C**. All other characteristic values are estimated to approximate typical production systems found in the literature. All design decisions for the generated controlled production systems are made randomly within the bounds specified by the input characteristics.

This arrangement allows the effects of individual production system characteristics to be tested independently. Also, any number of nearly identical production systems can be constructed and tested, providing accurate statistics. Every time or space cost appearing in tables below is a result of at least nine trials using at least three different, simulated controlled production systems.

5.4.2. Empirical Results

The first test investigates the worst case scenario of SECTION 5.2.2.1. Production system sizes range from 10 to 200 productions. Such worst case production systems must be control free, so **G** = **P**. All productions are legal all of the time. The results are presented in TABLE 5.1 below, where $\frac{msec}{fire}$ denotes average firing rate, and $\sigma(\frac{msec}{fire})$ denotes the standard deviation of that firing rate. All tests were performed on a PRIME 9950 minicomputer. As expected, the average rate of production firing (i.e., $\frac{msec}{fire}$) grows linearly with respect to **P**, as illustrated in FIGURE 5.6. The size of compressed CLR(0) parse tables (e.g., *size*) also grows linearly with respect to **P**.

One unexpected result is the apparent unimportance of the working memory size (**W**) on the firing rate. This result may be an artifact of the construction process. The worst case scenario is so unrealistic that a wide range of worst behaviors had to be constructed by the BUILDER program; one such behavior may be in error. More likely, the effects of **W** do not become noticeable until the ratio **W**/**P** becomes larger. Unfortunately, limitations of available parser generators precluded testing of the worst case scenario with large **W**.

The second test investigates the best case scenario of SECTION 5.2.2.2. The results of this investigation are summarized in TABLE 5.2. As expected, the firing rate appears constant with respect to **P** and **W**, as illustrated in FIGURE 5.7. Notice the difference in scale between the graphs of FIGURES 5.6 and 5.7.

The third test investigates the time and space costs of palimpsest parser implementations of typical controlled production systems. Two important theoretical results, predicted in SECTIONS 5.2, are supported by the data presented in TABLE 5.3. First, the firing rate of a controlled production system is constant with respect to the number of productions **P**; it is the module size **G** that determines the firing rate of a palimpsest parser. Second, this firing rate is very near the best case, as illustrated in FIGURE 5.8. These results for simulated controlled production systems, display a consistent firing rate of approximately 1000 productions per second.

				W = 10		W = 20	
P	C	G	Size	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$
10	5	10	598	7.71	±0.21	7.58	±0.22
25	5	25	1398	19.36	±0.52	19.28	±0.55
50	5	50	2718	38.49	±0.94	39.10	±1.07
75	5	75	4043	58.87	±1.59	62.12	±0.68
100	5	100	5368	79.64	±1.95	79.88	±1.11
125	5	125	6715	102.06	±2.41	101.45	±1.44
150	5	150	8040	132.24	±1.04	134.94	±2.50
175	5	175	9365	149.10	±3.53	144.73	±2.07
200	5	200	10690	162.46	±3.58	168.20	±2.00

TABLE 5.1. Worst Case Results

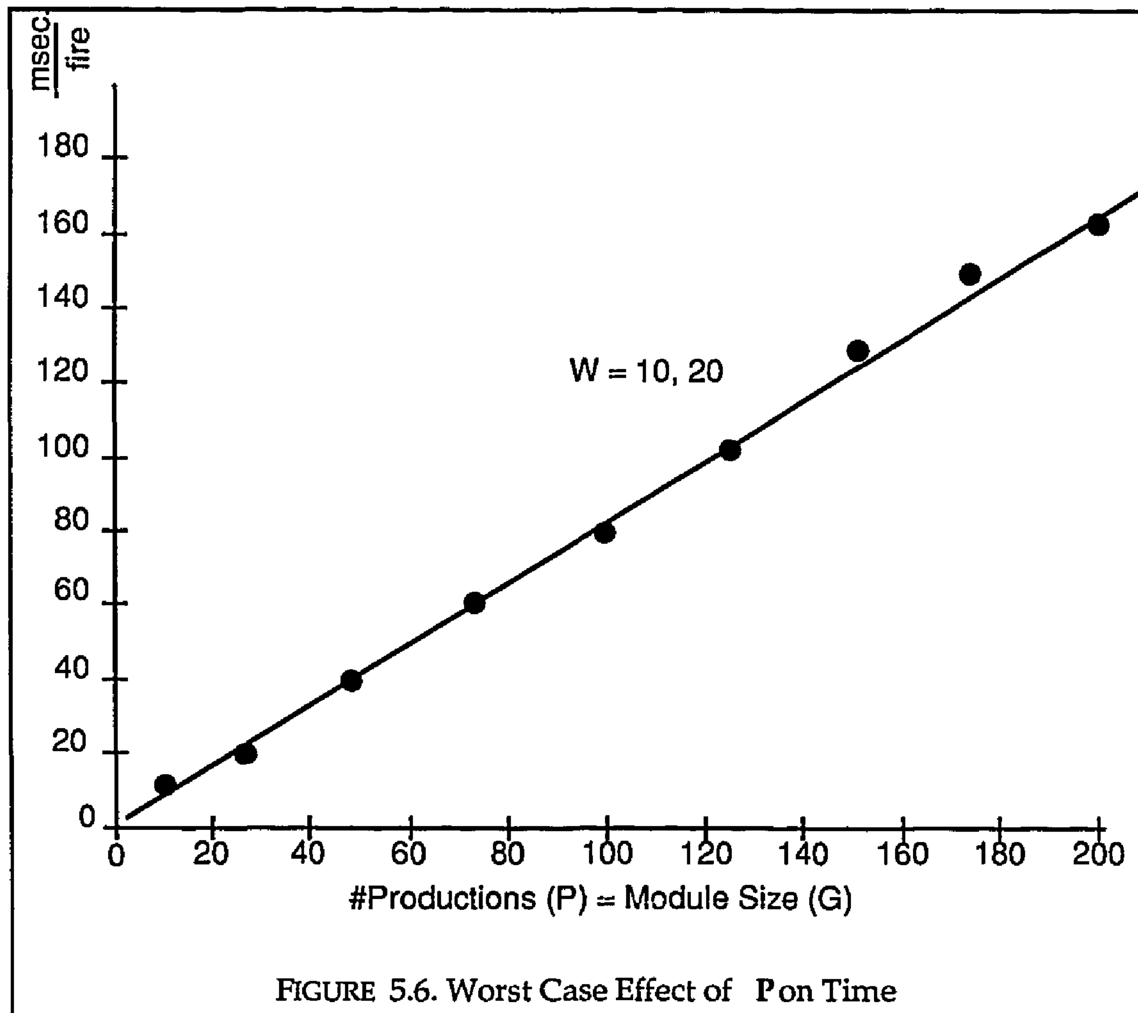


FIGURE 5.6. Worst Case Effect of P on Time

P	C	G	Size	W = 100 $\frac{msec}{fire}$	W = 100 $\sigma(\frac{msec}{fire})$	W = 200 $\frac{msec}{fire}$	W = 200 $\sigma(\frac{msec}{fire})$	W = 500 $\frac{msec}{fire}$	W = 500 $\sigma(\frac{msec}{fire})$
10	1	1	275	0.44	±0.02	0.44	±0.01	0.44	±0.01
25	1	1	627	0.45	±0.01	0.45	±0.01	0.45	±0.00
50	1	1	1199	0.42	±0.00	0.43	±0.01	0.43	±0.01
75	1	1	1782	0.43	±0.01	0.43	±0.02	0.44	±0.01
100	1	1	2354	0.44	±0.01	0.44	±0.01	0.45	±0.01
125	1	1	2937	0.44	±0.00	0.44	±0.01	0.44	±0.01
150	1	1	3509	0.47	±0.01	0.47	±0.01	0.47	±0.01
175	1	1	4092	0.48	±0.01	0.48	±0.01	0.48	±0.01
200	1	1	4664	0.45	±0.01	0.45	±0.01	0.45	±0.01

TABLE 5.2. Best Case Results

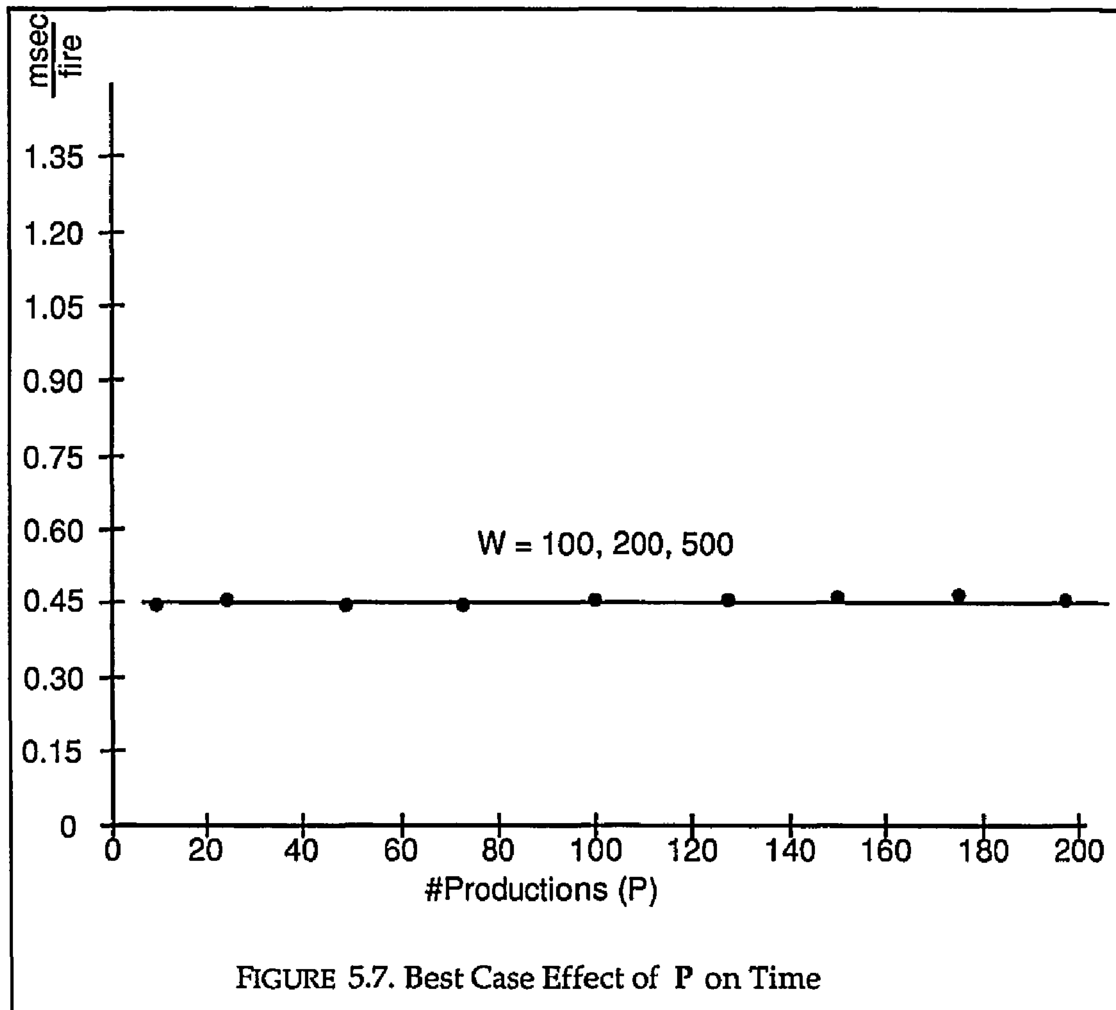


FIGURE 5.7. Best Case Effect of **P** on Time

				W = 100		W = 200		W = 500		W= 1000	
G	P	C	Size	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$	$\frac{msec}{fire}$	$\sigma(\frac{msec}{fire})$
3	25	3	879	1.12	±0.09	1.12	±0.08	1.12	±0.09	1.13	±0.08
3	50	3	1815	1.07	±0.01	1.06	±0.02	1.07	±0.02	1.07	±0.02
3	75	3	2694	0.95	±0.04	0.95	±0.05	0.95	±0.04	0.94	±0.04
3	100	3	3573	0.97	±0.04	0.99	±0.04	0.99	±0.02	0.99	±0.02
3	125	3	4509	1.06	±0.03	1.05	±0.03	1.06	±0.05	1.04	±0.06
3	150	3	5366	1.00	±0.01	1.00	±0.03	0.98	±0.04	1.01	±0.02
3	175	3	6278	0.97	±0.02	0.97	±0.03	0.94	±0.01	0.93	±0.05
3	200	3	7159	1.00	±0.05	0.97	±0.04	0.96	±0.03	0.95	±0.03
5	25	3	840	1.24	±0.08	1.25	±0.08	1.19	±0.08	1.22	±0.08
5	50	3	1680	1.09	±0.05	1.07	±0.05	1.10	±0.06	1.11	±0.01
5	75	3	2520	1.08	±0.04	1.09	±0.04	1.07	±0.03	1.04	±0.04
5	100	3	3349	0.97	±0.06	0.98	±0.06	0.98	±0.08	0.95	±0.07
5	125	3	4214	1.07	±0.05	1.07	±0.10	1.06	±0.10	1.02	±0.08
5	150	3	5051	1.06	±0.04	1.03	±0.04	1.04	±0.04	1.04	±0.03
5	175	3	5913	0.99	±0.06	1.00	±0.03	0.93	±0.03	0.94	±0.02
5	200	3	6753	0.99	±0.04	0.99	±0.03	0.98	±0.01	0.96	±0.01
10	25	3	781	1.38	±0.19	1.36	±0.15	1.36	±0.21	1.37	±0.23
10	50	3	1571	1.16	±0.13	1.17	±0.17	1.12	±0.16	1.13	±0.15
10	75	3	2341	1.08	±0.08	1.10	±0.10	1.07	±0.10	1.07	±0.11
10	100	3	3131	1.11	±0.04	1.09	±0.06	1.09	±0.08	1.06	±0.12
10	125	3	3945	1.08	±0.04	1.08	±0.03	1.07	±0.05	1.04	±0.02
10	150	3	4757	1.14	±0.06	1.14	±0.06	1.08	±0.05	1.05	±0.08
10	175	3	5571	1.07	±0.03	1.05	±0.05	1.05	±0.05	1.04	±0.08
10	200	3	6339	1.00	±0.06	1.02	±0.04	0.98	±0.06	0.95	±0.05
25	25	3	623	1.33	±0.22	1.31	±0.20	1.28	±0.19	1.23	±0.17
25	50	3	1326	1.40	±0.02	1.40	±0.03	1.41	±0.03	1.41	±0.04
25	75	3	2122	1.28	±0.03	1.27	±0.06	1.23	±0.08	1.19	±0.05
25	100	3	2844	1.22	±0.03	1.21	±0.05	1.19	±0.06	1.18	±0.06
25	125	3	3643	1.26	±0.06	1.25	±0.06	1.25	±0.04	1.23	±0.06
25	150	3	4431	1.15	±0.10	1.17	±0.10	1.12	±0.09	1.09	±0.11
25	175	3	5230	1.09	±0.04	1.09	±0.05	1.04	±0.03	1.00	±0.07
25	200	3	5908	1.13	±0.10	1.09	±0.07	1.07	±0.08	1.05	±0.07

TABLE 5.3. Typical Production System Results

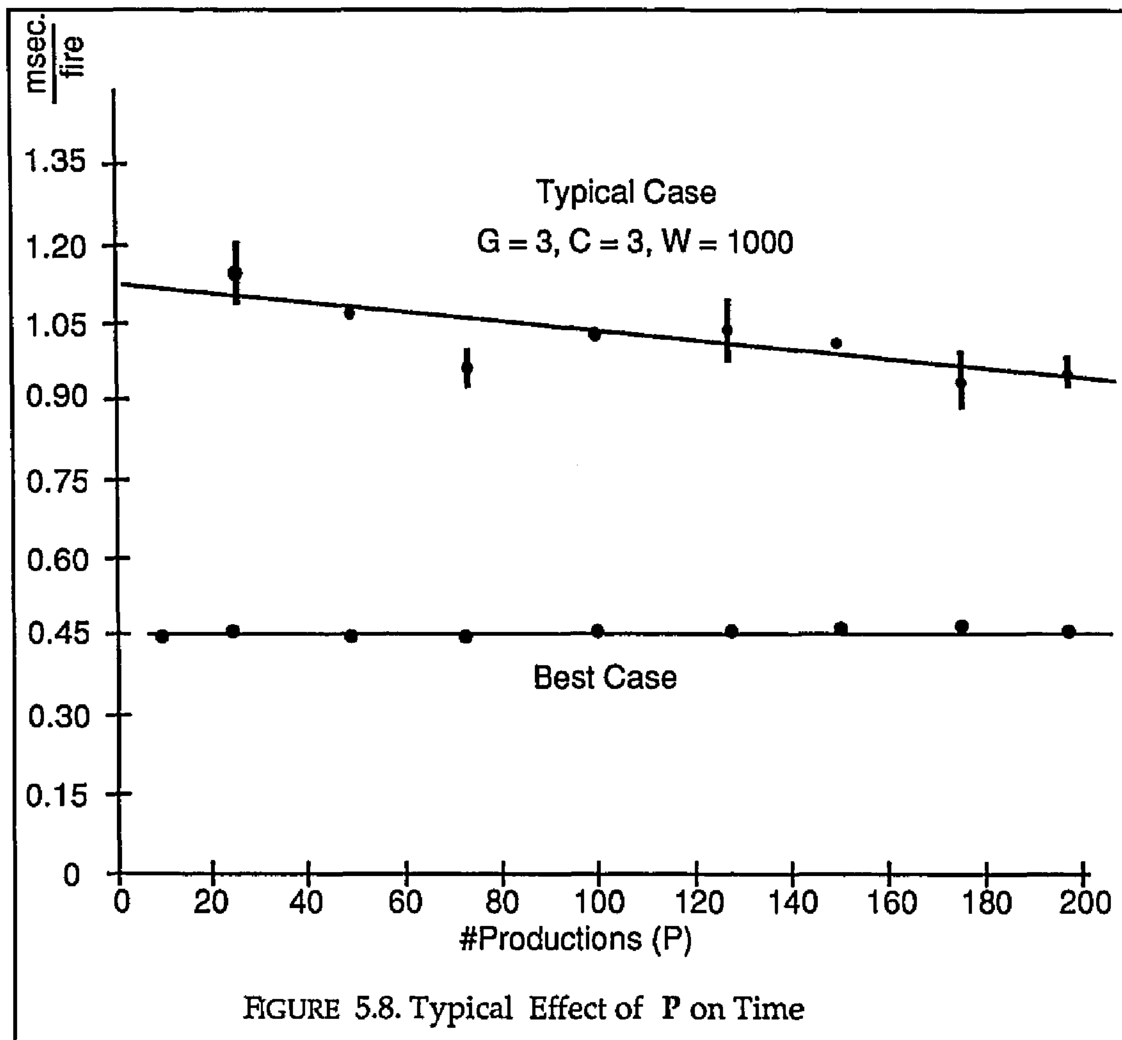


FIGURE 5.8. Typical Effect of P on Time

6. Palimpsest Parser Enhancements

As described above, palimpsest parsers meet the speed and modularity requirements of large production systems. Also, with little or no additional effort, three features that are also valuable in the large production system domain can be accommodated by the palimpsest parser architecture. This chapter presents brief descriptions of those features, and outlines their implementation where necessary. A number of directions for future research are then outlined.

6.1. Additional Features

This section describes three useful features of the palimpsest parser architecture.

6.1.1. Backward-Chaining Evaluation

The palimpsest parser production system architecture employs a forward-chaining evaluation strategy by looking for instantiations of legal productions and firing them as they are found. All goal structuring must be built into the control grammar. However, large production systems may contain productions or modules that deal with a very wide range of problems, some of which may be better suited to a backward-chaining evaluation. Fortunately, a minor change to the palimpsest transformation provides this capability[143].

First, the purpose of a backward chaining evaluation is to provide a goal directed search for instantiations. That is, suppose that a specific working memory element is required to satisfy a condition. For example, in the controlled production system of FIGURE 4.1, production ***loaditem*** needs an ele-

[143] Collins and Slothouber, 1988.

ment from the *OPENBAY* element class, as illustrated in the following palimpsest grammar rule:

LOADITEM $\rightarrow$ anyitem openbay

Also, suppose that another production exists that will create a desired element when it's action is applied, such as the production ***opennewbay***, represented by the following palimpsest grammar rule:

OPENNEWBAY $\rightarrow$ anyitem not_openbay closedbay

This situation might be represented by the production memory and control grammar of FIGURE 6.1, where all original occurrences of production ***opennewbay*** (now redundant) have been removed. To evaluate production ***opennewbay*** using backward chaining, first create a copy of the corresponding grammar rule, and change the LHS non-terminal symbol into the non-terminal form of the desired element class name. For example:

OPENBAY $\rightarrow$ anyitem not_openbay closedbay

Next, modify the goal rule by replacing the terminal form of the desired element class name by its non-terminal form:

LOADITEM $\rightarrow$ anyitem OPENBAY

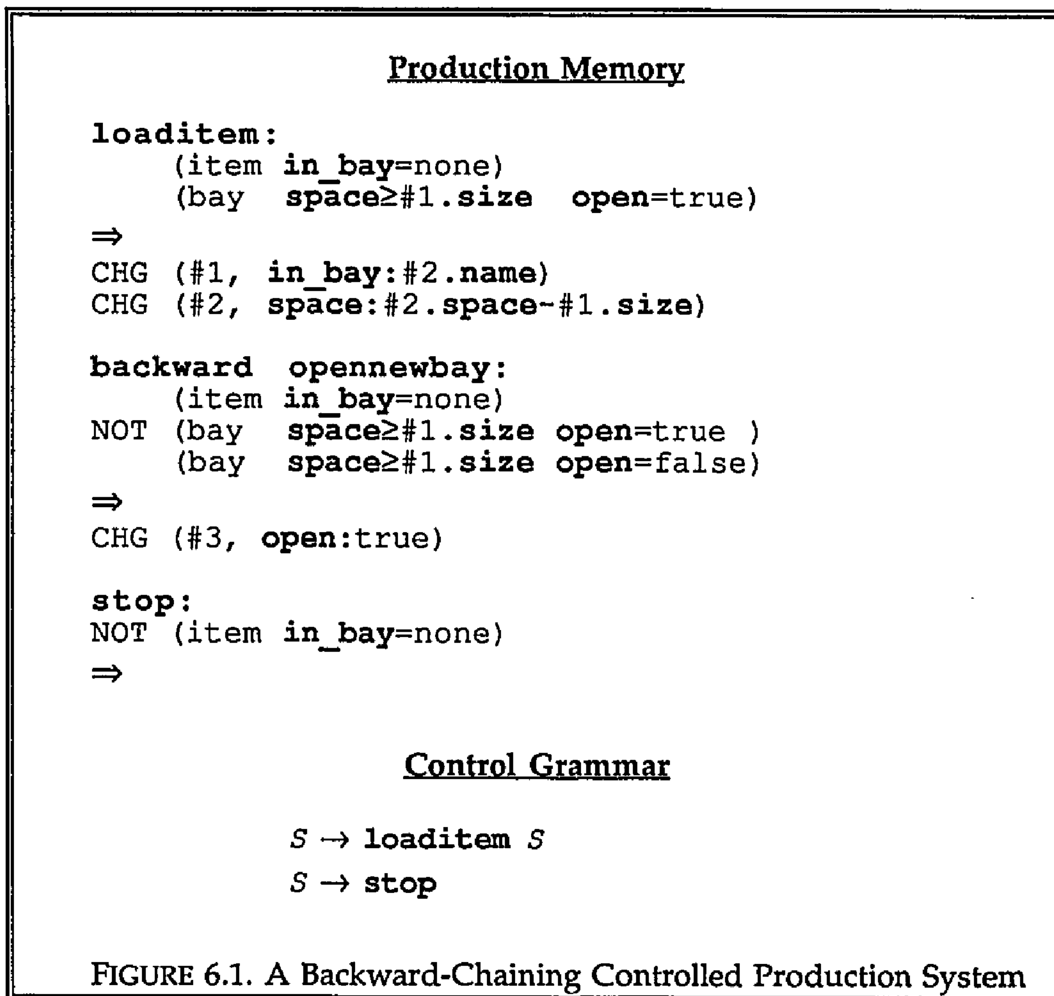
Finally, add a new grammar rule with a LHS containing the non-terminal form of the desired element class name, and a RHS containing its terminal form:

OPENBAY $\rightarrow$ openbay

For the examples above, the resulting palimpsest grammar would be

(1)	*S*	$\rightarrow$ LOADITEM *S*
(2)	*S*	$\rightarrow$ STOP
(3)	LOADITEM	$\rightarrow$ anyitem OPENBAY
(4)	STOP	$\rightarrow$ not_anyitem
(5)	OPENBAY	$\rightarrow$ anyitem not_openbay closedbay
(6)	OPENBAY	$\rightarrow$ openbay

If necessary, the implementation of the palimpsest parsing algorithm should be modified to treat both the non-terminal and terminal forms of the element



Production Memory

```
loaditem:
    (item in_bay=none)
    (bay  space≥#1.size  open=true)
⇒
CHG (#1, in_bay:#2.name)
CHG (#2, space:#2.space-#1.size)

backward  opennewbay:
    (item in_bay=none)
NOT (bay  space≥#1.size open=true )
    (bay  space≥#1.size open=false)
⇒
CHG (#3, open:true)

stop:
NOT (item in_bay=none)
⇒
```

Control Grammar

$S \rightarrow$ **loaditem** S

$S \rightarrow$ **stop**

FIGURE 6.1. A Backward-Chaining Controlled Production System

class name on the parse stack as if it were a terminal symbol; that is, allow backtracks over that symbol.

Execution of the resulting palimpsest parser will attempt to instantiate ***opennewbay*** (i.e., rule (5)) whenever an element from the *OPENBAY* element class is required by ***loaditem***. If ***opennewbay*** fires, an *OPENBAY* element is guaranteed to be created, and processing of ***loaditem*** continues. If ***opennewbay*** cannot be instantiated, the parser tries to apply rule (6), which guarantees that an *OPENBAY* element already exists. If neither rule (5) or rule(6) can be applied, the parser backtracks, just as if it had tried to shift ***openbay***.

Backward chaining rules can be used to instantiate other backward chaining rules, *ad infinitum*.

6.1.2. Separate Compilation Units

Many procedural languages, such as Ada, allow groups of modules to be written and compiled separately, facilitating top-down design of large programs. This situation can be emulated by palimpsest parsers[144]. First, compile every controlled production system representing a compilation unit into a palimpsest grammar, a disambiguation function, and a semantics function (including classification procedure). Care must be taken to insure that the same names are used for equivalent symbols in all compilation units. Combination of the separate compilation units is straightforward; palimpsest grammars are concatenated, the outermost **case** statements of each classification procedure are combined within one classification procedure skeleton, **case** statement alternatives for the various *d*'s and *r*'s are placed within a single procedure skeleton. A palimpsest parser is then constructed as usual from the combined palimpsest grammar, disambiguation function, and semantics function.

6.1.3. Conventional Conflict Resolution

The fire first selection strategy was chosen for palimpsest parsers to optimize performance. Should this selection strategy prove inadequate, then conflict resolution strategies such as *recency*, *specificity*, and *refraction*, and even full conventional conflict resolution may be incorporated into palimpsest parsers.

If the implementation of SECTION 5.1 is used, newly added or changed working memory elements are placed at the beginning of element class lists. Any instantiation found for a production will automatically contain the most recent working memory elements in the satisfying list. This is not exactly the *recency* conflict resolution function in the literature; but, if it is used for some purpose other than providing procedural control, this version of recency should be sufficient.

---

[144] Collins and Slothouber, 1988.

In the alternate form of CLR(0) parsing action functions, all shift actions appear before reduce actions. This means that given a choice between two similar, legal productions, a palimpsest parser will fire the one with the largest, most specific satisfied condition. This is not exactly the *specificity* conflict resolution function in the literature; but, if it is used for some purpose other than providing procedural control, this version of specificity should be sufficient.

In those rare instances where it is necessary, refraction can be implemented explicitly in any production system without resorting to full conventional conflict resolution. Add an attribute to the affected working memory elements. Such attributes are defined to have one value (or set of values) when the working memory element is allowed to be used in an instantiation. This attribute value is to be changed by a production's action to signify that the working memory element cannot be used to match the same pattern again.

If, for some unforeseeable reason, the fire first conflict resolution function is inadequate for some application, conventional conflict resolution can be performed by palimpsest parsers. Whenever an instantiation is found, the disambiguation predicate puts it in a conflict set (along with the current parse stack) and return *false*. Then, instead of firing an instantiation, the palimpsest parsing algorithm searches for other instantiations. After all instantiations are found, the palimpsest parser eventually blocks. At this point, instead of halting, conventional conflict resolution and act phases should be executed. The palimpsest parser then applies the proper parsing goto function for the production fired and begins the next cycle. The time penalty incurred by this process is quite high, but may be ameliorated with the use of conflict set support information[145].

145 Miranker, 1987.

6.2. Future Research

This section describes various directions for future research related to palimpsest parsers. These are preliminary ideas, and have not yet been investigated.

6.2.1. Modular Production System Language

Before palimpsest parsers can be used to interpret large controlled production systems, some large controlled production systems need to be written. Therefore, a modular production system language suitable for the construction of large production systems, and the transformation from that language to pure controlled production systems need to be defined and implemented.

6.2.2. Performance Optimizations

A number of approaches may lead to optimizations of palimpsest parser performance. First, careful analysis of the working memory elements created by production actions may allow the classification procedure (a potential bottleneck in large production systems) to be bypassed in many situations. Second, analysis of the relationships between productions and their relative ordering in the control sequences may allow redundant patterns and operations to be removed. Third, judicious re-ordering of symbols on the RHS of productions in a production memory grammar may allow condition membership to be used more effectively. Finally, a large body of performance related LR parsing research concerns grammar and parse table transformations. Much of this work may be applicable to palimpsest parsers.

6.2.3. Explanatory Capability

One useful feature provided by many production systems is the ability of the system to provide a trace of its reasoning. A parse tree, constructed during

the execution of a palimpsest parser, and annotated with satisfying lists and working memory changes, should provide all the necessary information. Such parse trees might be constructed explicitly by the production system actions, or implicitly by a modified palimpsest parsing algorithm. If the parse tree were represented as a structure in working memory, the trace information would be available to the production system itself.

6.2.4. Uncertainty

Many production system architectures provide the capability to deal with uncertain information. For example, certainty factors may be associated with every working memory element that define the likelihood that the working memory element is correct. Such a scheme might be emulated by palimpsest parsers in one of two ways. The production system programmer may wish to explicitly put a certainty factor attribute within all working memory elements, and have productions examine and modify that information. Another approach is to implicitly associate certainty factors with every working memory element. The palimpsest disambiguation and semantics functions could then be modified to automatically maintain uncertainty information and use that information in the match process.

6.2.5. Reasoning About Controlled Production Systems

One potential problem in the construction of large production systems is that many productions may be redundant, inconsistent, unreachable or even unsatisfiable. The palimpsest parser architecture may allow many of these problems may be spotted upon examination of the palimpsest grammar for a controlled production system. For example, redundant productions will result in palimpsest grammar rules with the same or similar RHSs. Inconsistent and incorrect productions often result in palimpsest grammar rules that reference new and unexpected element classes.

6.2.6. Concurrency

One often mentioned, but seldom implemented feature of many production system architectures is the ability to interpret different productions, or groups of productions concurrently. Palimpsest parsers may be able interpret concurrent productions and production modules. One approach is to use a separate parse stack for every concurrent module, and modify the palimpsest parsing algorithm to cycle among the tasks applying one parse action for each task on each cycle.

7. Conclusion

It has been argued that *"programming in the large* is an essentially distinct and different intellectual activity from that of constructing individual modules... [and] ... essentially distinct and different languages should be used for the two activities."[146] A host of production system languages exist that are suitable for creating small applications; yet none provide a truly modular environment for programming in the large. Furthermore, the speed of conventional production system architectures that support these languages are prohibitively slow for large production systems. This thesis introduces a new production system architecture, called the palimpsest parser, that adapts LR parsing technology to the process of interpreting large controlled production systems. Controlled production systems provide a formal foundation upon which to design modular production system languages for programming in the large; and palimpsest parsers exploit that modular structure to interpret production systems fast, regardless of size.

Controlled production systems are compiled into palimpsest parsers as follows. Initially, the *palimpsest transformation* is applied to all productions to transform them into context-free grammar rules with associated disambiguation predicates and semantics. This grammar and the control grammar are then concatenated and compiled into modified LR(0) parse tables using conventional parser generation techniques. The resulting parse tables, disambiguation predicates, and semantics, in conjunction with a backtracking LR(0) parsing algorithm, constitute a palimpsest parser. When executed, this palimpsest parser correctly interprets the original controlled production system. Moreover, on any given cycle, the palimpsest parser only attempts to instantiate those productions that are allowed to fire by the control language grammar. Tests of simulated production systems[147] have consistently

[146] DeRemer and Kron, 1976, pp. 80-86.

[147] Simulated production systems had the following characteristics: 200 productions, 500 working memory elements, an average of 3 patterns and 3 actions per production.

exhibited firing rates in excess of 1000 productions per second on a conventional minicomputer.

Bibliography

Aho, A. V., and Ullman, J. D. (1972) *The Theory of Parsing, Translation, and Compiling,* Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey.

Allen, L., (1982) "YAPS: Yet another production system," Technical Report TR-1146, Department of Computer Science, University of Maryland.

AT&T Information Systems (1986) *The UNIX System User's Manual,* Prentice-Hall, Englewood Cliffs, New Jersey.

Barker, V. E., and O'Connor, D. E., (1989) "Expert systems for configuration at Digital: XCON and beyond," *Communications of the ACM,* Vol. 32, #3, 298-318.

Brainerd, W. S., and Landweber, L. H. (1974) *Theory of Computation,* Wiley & Sons, New York, New York.

Brownston, L., Farrell, R., Kant, E., and Martin N., (1985) *Programming expert systems in OPS5,* Addison-Wesley, Reading, Mass.

Buchanan, B. G., and Shortliffe, E. H., (1984) *Rule Based Expert Systems,* Addison-Wesley, Reading, Mass.

Clocksin, W. F., and Mellish, C. S., *Programming in PROLOG,* Springer-Verlag, New York.

Collins, W.R., and Feyock, S., (1985) "The MYSTRO SYSTEM: the use of parsing technology to build real-time expert systems," Department of Computer Science, College of William and Mary, Final Report, Langley Research Center Grant NAG-1-469.

Collins, W.R., and Noonan, R.E., (1985) "The MYSTRO SYSTEM: a comprehensive translator toolkit," Department of Computer Science, College of William and Mary, Final Report, Langley Research Center Grant NSG-1435.

Collins, W.R., and Slothouber, L.P., (1988) "Expert system control in Ada," *Proc. of the 4th Annual Artificial Intelligence in Ada Conference*, MacLean, VA, 17.1-17.11

Davis, R., and King, J. (1976) "An overview of production systems," in *Machine Intelligence*, Vol. 8 (Elcock, E. W., and Michie, D., eds.), Wiley & Sons, New York, 300-332.

Davis, R., and Buchanan, B. G. (1977) "Meta level knowledge: Overview and implications," *Proc. 5th Int. Joint Conf. Artificial Intelligence*, Cambridge, Massachusetts, 920-927.

Davis, R. (1980) "Meta-rules: reasoning about control," in *Machine Intelligence*, Vol. 8 (Elcock, E. W., and Michie, D., eds.), Wiley & Sons, New York, 300-332.

Dencker, P., Dürre, K., and Heuft, J. (1984) "Optimization of parser tables for portable compilers," *ACM Transactions on Programming Languages and Systems*, Vol. 6, #4, 546-572.

DeRemer, F., and Kron, H. (1976) "Programming-in-the-large versus programming-in-the-small," *IEEE Transactions on Software Engineering SE-2*.

Duda, R. O., Hart, P. E., Konolige, K., and Reboh, R., (1979) "A computer-based consultant for mineral exploration," Technical Report, Final Report, SRI Project 6415, SRI International.

Feyock, S., (1984) "Syntax programming," *American Association for Artificial Intelligence National Conference*, Austin, Texas.

Floyd, R. W. (1961) "A descriptive language for symbol manipulation," *Journal of the ACM*, Vol. 8, #4, 579-584.

Forgy, C. L. (1979) "On the efficient implementation of production systems," Ph.D. Thesis, Carnegie-Mellon University, 1979.

Forgy, C. L. (1982) "Rete: a fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, Vol. 19, 17-37.

Forgy, C. L. (1984) "The OPS83 report," Technical Report CMU-CS-81-133, Department of Computer Science, Carnegie-Mellon University.

Forgy, C. L., Gupta, A., Newell, A., Wedig, R., (1984) "Initial assessment of architectures for production systems," *Proc. American Association for Artificial Intelligence.*

Forgy, C. L., Gupta, A., Newell, A., Wedig, R., (1986) "Parallel algorithms and architectures for rule-based systems," *Proc. 13th Annual Int. Symp. on Computer Architecture*, Tokyo, Japan, 28-37.

Georgeff, M. P., (1982) "Procedural control in production systems," *Artificial Intelligence*, Vol. 18, 175-201.

Ghedi C., and Jazayeri, M., (1982) *Programming Language Concepts*, Wiley & Sons, Inc., New York.

Giarratano, J. C., (1988) "CLIPS User's Guide," Artificial Intelligence Section, Lyndon B. Johnson Space Center, Houston, Texas.

Hardy, G. H., Littlewood, J. E., and Pólya, G. (1959) *Inequalities, 2nd Edition*, Cambridge University Press, Cambridge.

Jacob, J. K., and Froscher, J. N., (1986) "Developing a software engineering methodology for knowledge-based systems," NRL report 9019, Naval Research Laboratory, Washington, D. C.

Jacon, J. K., and Froscher, J. N., (1985) "Designing expert systems for ease of change," *Proc. IEEE Expert Systems in Government Symposium*, McLean Virginia.

Kowalski, R., (1979) "Algorithm = logic + control," *Communications of the ACM*, Vol. 22, #7, 424-436.

McDermott, J. and Forgy, C. (1977) "Production system conflict resolution strategies," in *Pattern-Directed Inference Systems,* (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, London, 177-199.

McDermott, J., Newell, A., and Moore J. (1978) "The efficiency of certain production system implementations," in *Pattern-Directed Inference Systems,* (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, London, 155-176.

McDermott, J. (1980) "R1: a rule-based configurer of computer systems," Technical Report CMU-CS-80-119, Carnegie-Mellon University.

Minsky, M. (1967) *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, New Jersey.

Miranker, D. P. (1987) "TREAT: A new and efficient match algorithm for AI production pystems," Technical Report, Department of Computer Science, University of Texas at Austin.

Nuutila, E., Kuusela, J., Tamminen, M., Veilahti, J. Arkko, J. Bouteldja, N. (1987) "XC - a language for embedded rule based systems," *SIPGLAN NOTICES,* Vol. 22, #9, 23-32.

Olflazer, K., "Partitioning in parallel processing of production systems," Ph. D. Thesis, Carnegie Mellon University.

Post, E. (1943) "Formal reductions of the general combinatorial problem," *American Journal of Math.,* Vol. 65, 197-268.

Rhyne, J. R. (1977) "On finding conflict sets in production systems," Technical Report UH-CS-77-5, University of Houston.

Rieger, C. (1978) "Spontaneous computation and its roles in AI modeling," in *Pattern-Directed Inference Systems,* (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, London, 69-97.

Rychener, M. D. (1977) "Control requirements for the design of production system architectures," *SIGPLAN/SIGART Newsletter,* 37-44.

Savitch, W. J. (1982) *Abstract Machines and Grammars,* Little, Brown and Co., Boston, Mass.

Shortliffe, E. H. (1976) *Computer based medical consultations: MYCIN,* American Elsevier, New York.

Stefik, M., Aikens, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., and Sacerdoti, E. D. (1982) "The organization of expert systems," *Artificial Intelligence,* Vol. 18, 135-173.

Stolfo, S. J., and Shaw, D. E. (1982) "DADO: a tree-structured machine architecture for production systems," *Proc. National Conference on Artificial Intelligence, AAAI-1982.*

Waterman, D. A., and Hayes-Roth, F. (1978) "An overview of pattern-directed inference systems," in *Pattern-Directed Inference Systems,* (Waterman, D. A., and Hayes-Roth, F., eds.), Academic Press, London, 3-22.

Weiss, S. M., and Kulikowski, C. A., (1981) "Expert consultation systems: The EXPERT and CASNET projects," in *Machine Intelligence,* Infotech State of the Art Report 9, #3, Pergamon Infotech Ltd., Maidenhead Berks, England.

Appendix A. Notation

Throughout this thesis, it is assumed that the reader is familiar with the following notation and definitions:

- $\equiv$ is mathematical shorthand for *"identical with."*
- $\forall$ is the *universal quantifier,* and is read as "for all."
- $\exists$ is the *existential quantifier,* and is read as "there exists."
- $\ni$ is mathematical shorthand for *"such that."*
- *iff* is mathematical shorthand for *"if and only if."*
- $\subseteq$ is the *subset* operator.
- $\in$ is the *set membership* operator.
- $\cap$ is the *set intersection* operator.
- $\cup$ is the *set union* operator.
- An *alphabet* is a non-empty, finite set of primitive symbols.
- A *word* in Σ is a finite string of symbols from alphabet Σ.
- The *length* of a word α, denoted $|\alpha|$, is the number of symbols in α.
- The *empty word,* denoted ε, is a word of length 0.
- Σ^{*k} is the set of all words in Σ with length less than or equal to k.
- Σ^{+} is the set of all words in Σ of length 1 or greater.
- $\Sigma^{*} \equiv \Sigma^{+} \cup \{\varepsilon\}$.
- $\alpha_{x,y}$ denotes the list of items $\langle \alpha_x, \alpha_{x+1}, \dots, \alpha_y \rangle$.
- 2^{Φ} is the *powerset* of the set Φ; that is, $\{\phi \mid \phi \subseteq \Phi\}$.
- $\Phi \times \Psi$ denotes the *cartesian product* of sets Φ and Ψ.
- $\Phi^k \equiv \Phi \times \Phi \times \dots k\text{-times} \dots \times \Phi$.
- $\Phi^* \equiv \Phi^1 \cup \Phi^2 \cup \Phi^3 \cup \dots$
- $\Phi \sim \Psi$ denotes the *set difference* of sets Φ and Ψ.
- $\wedge$ is the logical *AND* operator.
- $\vee$ is the logical *OR* operator.
- $\Rightarrow$ separates a production's condition and action.
- $\overset{*}{\underset{rm}{\Rightarrow}}$ denotes *"is a rightmost derivation of."*
- $\square$ marks the end of examples, definitions, algorithms, and proofs.

Also, many definitions reference notational abbreviations that have been introduced prior to that definition. A list of these abbreviations follows along with the page number on which they first appear. Subscripted forms of

abbreviations (e.g., w_i is the subscripted form of w) are not shown. Occasionally, some of these abbreviations are overloaded, but the meanings should be clear from context.

Page	Abbrev.	Meaning
9	**P**	The number of productions in a production memory.
9	**W**	The number of data elements in a working memory.
9	$\mathbf{C_i}$	The number of patterns in production number i.
9	$\mathbf{C_i^+}$	The number of positive patterns in production i.
9	$\mathbf{C_i^-}$	The number of negative patterns in production i.
21	Σ	An alphabet of primitve symbols in a PPS.
21	Ψ	An alphabet of variables in a PPS.
22	G	A context-free grammar (CFG).
22	$L(G)$	The language defined by a CFG (G).
31	Φ	A production system alphabet.
31	U	A working memory element universe; $U \subseteq \Phi^+$.
31	$C^{(i)}$	A category in U; $U = \{C^{(1)}, \ldots, C^{(n)}\}$.
31	C	A category in U; $C \in \{C^{(1)}, \ldots, C^{(n)}\}$.
31	V_i	A set of values.
31	A	An attribute of some category C; $A : C \rightarrow V_i$.
31	W	A working memory; $W \subseteq U$.
31	w	A working memory element; $w \in W$.
33	U^k	All lists of working memory elements in U.
33	p_i^k	A k-predicate; $p_i^k : C_1 \times \ldots \times C_k \rightarrow \{true, false\}$.
35	$p_i^{k'}$	The one-element component of p_i^k.
35	$p_i^{k''}$	The many-element component of p_i^k.
35	$w_{1,k}$	The list $\langle w_1, w_2, \ldots, w_k \rangle$.
36	$p_i^k\downarrow$	The q_1^1 such that $q_1^1(w_i) = p_i^{k'}(w_{1,k})$.
37	$\langle +, p_i^k \rangle$	A positive k-pattern.
37	$\langle -, p_i^k \rangle$	A negative k-pattern.
37	$\langle s_i, p_i^k \rangle$	A k-pattern of arbitrary sign.
37	c	A category; $c = \{\langle s_1, p_1^k \rangle, \ldots, \langle s_k, p_k^k \rangle\}$.
38	2^U	The set of all working memories of U.
38	o	An operation; $o : C_1 \times \ldots \times C_k \times 2^U \rightarrow 2^U$.
39	a	An action; $a(w_{1,n}, W) = o_m(w_{1,n}, o_{m-1}(w_{1,n}, .. o_1(w_{1,n}, W) ..))$.

40	n	A production name; $n \in \Phi^+$.
40	$\langle n, c, a \rangle$	A production on U.
40	$\mathbb{P}(U)$	The set of all productions on U.
40	$\mathcal{P}$	A production memory on U; $\mathcal{P} \subseteq \mathbb{P}(U)$.
40	$\langle \mathcal{P}, W \rangle$	A production system on U.
43	$\mathbb{C}(U)$	The set of all instantiations on $\langle \mathbb{P}(U), U \rangle$.
43	M	A match function; $M : 2^{\mathbb{P}(U)} \times 2^U \rightarrow 2^{\mathbb{C}(U)}$.
43	R	A conflict resolution function; $R : 2^{\mathbb{C}(U)} \rightarrow \mathbb{C}(U)$.
43	$\langle M, R \rangle$	A selection strategy.
45	$N(\mathcal{P})$	The set of all production names found in $\mathcal{P}$.
45	K	A control language of $\mathcal{P}$; $K \subseteq N(\mathcal{P})^*$.
45	G_K	A control grammar for control language K.
45	$(\langle \mathcal{P}, W \rangle, G_K)$	A controlled production system.
57	$\mathcal{S}$	A set of LR(0) items.
57	a	An LR(0) item set.
57	γ	A viable prefix.
57	$V(\gamma)$	The item set for a viable prefix γ.
58	P	The set of grammar rules in a CFG.
58	S	The start symbol in a CFG.
58	f	A parsing action function.
58	g	A parsing goto function.
58	$T(a)$	An LR(0) parse table for a.
61	$\mathcal{F}$	A parsing function.
62	f_C	A CLR(0) parsing action function.
62	$\mathcal{A}$	The set of all parsing actions in a set of parse tables.
66	d	A (palimpsest) disambiguation function.
66	r	A (palimpsest) semantics (or reduce semtantics) function.
87	$\langle id, p \rangle$	An element class table entry.
87	p	An element class specification.
87	$\mathcal{E}$	An element class table.
88	$G_{\mathcal{P}}$	A production memory grammar for $\mathcal{P}$.
89	$G_{\mathcal{P}K}$	A palimpsest grammar.
90	U^*	The set of all lists of working memory elements in U.

Appendix B. A Complete Production System Example

This appendix describes the design and transformation process for a complete, albeit small, production system. The first section describes the top-down design of the production system from the problem specification. The next section presents the element class table, palimpsest grammar, disambiguation function and semantics generated from the production system by the palimpsest transformation. Finally, a sample execution trace for the resulting palimpsest parser is presented.

B.1. Designing a Production System

The goal is to design a production system, called *LOADBAY*, that finds a near optimal solution, much as a human might, to the following problem.

Determine how to load a list of items into a minimum number of cargo bays, subject to the following constraints:

- *No flammable fuel may be loaded into cargo bays.*
- *At least one food item must be loaded.*
- *No two items of the same type may be loaded into the same bay.*

It is assumed that enough cargo space exists to load all items.

The inventory of items and the bays into which they are to be loaded will be represented by a working memory, as in FIGURE B.1.

The constraints suggest that the problem can be decomposed into three distinct phases: removing all fuel items, adding one food item, and loading cargo bays. This decomposition may be represented by the following control grammar rule, where non-terminals are surrounded by "<" and ">":

<loadbay> → <remove_all_fuel> <add_one_food> <load_bays>

```
(item  name:item1  type:water  size:67   bay:none)
(item  name:item2  type:fuel   size:98   bay:none)
(item  name:item3  type:air    size:83   bay:none)
(item  name:item4  type:suit   size:77   bay:none)
(item  name:item5  type:scope  size:71   bay:none)
(item  name:item6  type:hose   size:95   bay:none)
(item  name:item7  type:book   size:53   bay:none)
(item  name:item8  type:wrench size:39   bay:none)
(item  name:item9  type:paper  size:34   bay:none)
(item  name:item10 type:pen    size:19   bay:none)
(item  name:item11 type:nut    size:10   bay:none)
(item  name:item12 type:bolt   size:9    bay:none)
(item  name:item13 type:fuel   size:71   bay:none)
(item  name:item14 type:camera size:47   bay:none)
(bay   name:bay1   size:50     space:50  open:false)
(bay   name:bay2   size:150    space:150 open:false)
(bay   name:bay3   size:300    space:300 open:false)
(bay   name:bay4   size:100    space:100 open:false)
(bay   name:bay5   size:100    space:100 open:false)
```

FIGURE B.1. An Initial *LOADBAY* Working Memory

Removing fuel requires a loop, because more than one fuel item may be in working memory. This loop may be represented by the following control grammar rules, where terminal symbols represent production names:

<remove_all_fuel> → remove_fuel <remove_all_fuel>

<remove_all_fuel> → ε

The ***remove_fuel*** production will be repeatedly applied until it's condition is no longer satisfied.

Food is to be added only if none already exists in the inventory. Such an optional production application is represented by the following control grammar rules:

<add_one_food> → add_food

<add_one_food> → ε

In order to load bays as full as possible, a human might apply the following two strategies:

Strategy A

- *Load larger items before smaller items.* Items will be divided into three classes: *large* (i.e., $size \geq 66$), *medium* (i.e., $33 \leq size < 66$), and *small* (i.e., $size < 33$).
- *Only open new bays when no more items fit into old ones.* It is assumed that no bays are initially open.

Strategy B

- *Move items between bays to free larger blocks of space.* Circular moves are avoided if items are only moved into a destination bay with less space than the origin bay.
- *Swap items between bays to free larger blocks of space.* Circular swaps are avoided if the larger of the two items is swapped into the bay with the lesser space.

Strategy A is designed to provide an approximate initial loading of items into bays. Strategy B is designed to try to optimize the loading performed by Strategy A. How these two strategies are to be applied? Strategy B must be applied after Strategy A so that cargo bays will have loaded items to move around. But how often should Strategy B be applied? every time Strategy A is applied? or once, after all items have been loaded? One approach is to apply Strategy B only if the amount of wasted space in any bay exceeds some acceptable value, say, 15 percent. This approach may be represented in the control grammar of FIGURE B.2, in which Strategy A and Strategy B are to be applied in a loop until all items have been loaded. Strategy B is applied only if it has not already been applied to the current loading scheme and if a bay exists with more than 15 percent wasted space. Otherwise, Strategy A is applied.

<loadbay>	→ <remove_all_fuel> <add_one_food>
+	<load_bays>
<remove_all_fuel>	→ remove_fuel <remove_all_fuel>
<remove_all_fuel>	→ ε
<add_one_food>	→ add_food
<add_one_food>	→ ε
<load_bays>	→ check_load <load_loop>
<load_loop>	→ <load_loop> <strategy_a>
<load_loop>	→ <load_loop> <strategy_b>
<load_loop>	→ ε
<strategy_a>	→ check_a1 <open> <load_big> <load_med>
+	<load_small>
<strategy_a>	→ check_a2 <open> <load_big> <load_med>
+	<load_small>
<open>	→ **open_new_bay**
<open>	→ ε
<load_big>	→ **load_big_item** <load_big>
<load_big>	→ ε
<load_med>	→ **load_med_item** <load_med>
<load_med>	→ ε
<load_small>	→ **load_small_item** <load_small>
<load_small>	→ ε
<strategy_b>	→ check_b <strategy_b_loop>
<strategy_b_loop>	→ **swap_items** <strategy_b_loop>
<strategy_b_loop>	→ **move_item** <strategy_b_loop>
<strategy_b_loop>	→ ε

FIGURE B.2. The *LOADBAY* Control Grammar[148]

A number of productions (terminal symbols) were referenced in the control grammar. Each production is supposed to perform a single, independent, well-defined function. The top-down design process described above has insured that this is so. Their definition, given their use in the control grammar and the original problem specification, is straightforward. FIGURE B.3 lists these productions. Notice the last four *"check_"* productions are used to "guard" specific strategies and signal loop termination.

---

148 A "+" symbol in colum one indicates a continuation of the previous rule.

```
remove_fuel:
    (item  type=fuel)
⇒
REM (#1)

add_food:
NOT (item  type=food)
⇒
ADD (item name:genname(item)  type:food  size:10
in_bay:none)

open_new_bay:
    (item in_bay=none)
NOT (bay  space≥#1.size  open=true )
    (bay  space≥#1.size  open=false)
⇒
CHG (#3, open:true)

load_large_item:
    (item size≥66  in_bay=none)
    (bay  space≥#1.size  open=true)
NOT (item type=#1.type  in_bay=#2.in_bay)
⇒
CHG (#1, in_bay:#2.name)
CHG (#2, space:(#2.space-#1.size) )

load_med_item:
    (item size≥33  size<66  in_bay=none)
    (bay  space≥#1.size  open=true)
NOT (item type=#1.type  in_bay=#2.in_bay)
⇒
CHG (#1, in_bay:#2.name)
CHG (#2, space:(#2.space-#1.size) )

load_small_item:
    (item size<33  in_bay=none)
    (bay  space≥#1.size  open=true)
NOT (item type=#1.type  in_bay=#2.in_bay)
⇒
CHG (#1, in_bay:#2.name)
CHG (#2, space:(#2.space-#1.size) )

move_item:
    (bay  open=true)
    (bay  space>#1.space  open=true)
    (item size≤#1.space  in_bay=#2.name)
NOT (item in_bay=#1.in_bay  type=#3.type)
⇒
CHG (#3, in_bay:#1.name)
CHG (#2, space:(#2.space-#3.size) )
CHG (#1, space:(#1.space+#3.size) )
```

FIGURE B.3. (part 1) The *LOADBAY* Productions

```
swap_items:
    (bay   open=true)
    (bay   space>#1.space   open=true)
    (item  size<#4.size                in_bay=#1.name)
    (item  size≤(#1.space+#3.size)   in_bay=#2.name)
NOT (item  name≠#3.name   in_bay=#3.in_bay   type=#4.type)
NOT (item  name≠#4.name   in_bay=#4.in_bay   type=#3.type)
⇒
CHG (#4,  in_bay:#1.name)
CHG (#3,  in_bay:#2.name)
CHG (#2,  space:(#2.space-#4.size+#3.size) )
CHG (#1,  space:(#1.space-#3.size+#4.size) )

check_load:
⇒
ADD (strategy  swapped:true)

check_a1:
    (strategy  swapped:true)
⇒

check_a2:
    (item  in_bay=none)
NOT (bay    (space / size) > 0.15  open=true)
⇒

check_b:
    (strategy swapped=false)
    (bay     (space / size) > 0.15  open=true)
⇒
CHG (#1, swapped:true)
```

FIGURE B.3. (part 2) The *LOADBAY* Productions

Productions ***load_big_item, load_med_item,*** and ***load_small_item*** put items into open bays that have enough space to store big, medium, and small items, respectively; no two items of the same type are loaded into the same bay. The productions ***move_item*** and ***swap_items*** try to reduce wasted space by moving loaded items between bays. The productions ***add_food*** and ***remove_fuel*** add food and remove fuel, respectively. Lastly, the production ***open_new_bay*** opens a closed bay if one exists, and if there are unloaded items that do not fit into any already open bay. These productions do not specify any procedural control; this is done entirely by the control grammar in FIGURE B.2.

B.2. Applying The Palimpsest Transformation

The first step of the palimpsest transformation is to determine the element classes represented by the patterns in all productions. FIGURE B.4 presents the element classes represented by the *LOADBAY* production system above. In the implementation of SECTION 5.1, an element class table is only used internally by the palimpsest transformation; a classification procedure, that evaluates the element class specification predicates on working memory elements, is inserted into a palimpsest parser. The optimizations described in SECTION 5.1.3. are applied to this procedure to reduce redundant and unnecessary predicate testing. A classification procedure for the *LOADBAY* production system is shown in FIGURE B.5. Given a classification procedure, the palimpsest transformation produces a palimpsest grammar rule for every object-level production and control grammar rule. In addition, disambiguation, semantics functions are defined, and a portion of each is associated with every generated palimpsest grammar rule.

Element Class	Specification Predicate
LARGEITEM	$w \in$ item $\wedge$ $size(w) \geq 66$ $\wedge$ $in_bay(w)$ = none
OPENBAY	$w \in$ bay $\wedge$ $open(w)$ = true
ITEM	$w \in$ item
MEDITEM	$w \in$ item $\wedge$ $33 \leq size(w) \geq 66$ $\wedge$ $in_bay(w)$ = none
SMALLITEM	$w \in$ item $\wedge$ $size(w) < 33$ $\wedge$ $in_bay(w)$ = none
FOOD	$w \in$ item $\wedge$ $type(w)$ = food
FUEL	$w \in$ item $\wedge$ $type(w)$ = fuel
ANYITEM	$w \in$ item $\wedge$ $in_bay(w)$ = none
CLOSEDBAY	$w \in$ bay $\wedge$ $open(w)$ = false
SWAPPED	$w \in$ strategy $\wedge$ $swapped(w)$ = true
UNSWAPPED	$w \in$ strategy $\wedge$ $swapped(w)$ = false
WASTEDSPACE	$w \in$ bay $\wedge$ $open(w)$ = true $\wedge$ $(size(w) / space(w)) > 0.15$

FIGURE B.4. *LOADBAY* Element Class Table

```
procedure classify (w : working_memory_element) is
      prev : member_ptr;
begin
   prev := null;
   case w.category is
      when item =>
            add_to_elt_class_list(ITEM, w, prev);
            case w.type is
               when food =>
                  add_to_elt_class_list(FOOD, w, prev);
               when fuel =>
                  add_to_elt_class_list(FUEL, w, prev);
               when others =>
                  if w.in_bay = none then
                     if w.size < 33 then
                        add_to_elt_class_list(LARGEITEM, w, prev);
                     elseif w.size ≥ 66 then
                        add_to_elt_class_list(SMALLITEM, w, prev);
                     else
                        add_to_elt_class_list(MEDITEM, w, prev);
                     endif;
                  end if;
            end case;
      when bay =>
         if w.open then
            add_to_elt_class_list(OPENBAY, w, prev);
            if (w.size / w.space) > 0.15 then
               add_to_elt_class_list(WASTEDSPACE, w, prev);
            end if;
         else
            add_to_elt_class_list(CLOSEDBAY, w, prev);
         end if;
      when strategy =>
         if w.swapped then
            add_to_elt_class_list(SWAPPED, w, prev)
         else
            add_to_elt_class_list(UNSWAPPED, w, prev);
         end if;
   end case;
end classify;
```

FIGURE B.5. The *LOADBAY* Classification Procedure

The resulting palimpsest grammar for *LOADBAY* is shown starting on page 154. Note:

- Lines that begin with a "*" are comments.
- Lines that begin with a "<" are palimpsest grammar rules.
- Lines that begin with a "/" are chunks of disambiguation code associated with the most recent grammar rule.

- Non-blank lines that begin with a blank are chunks of semantics code associated with the most recent grammar rule.
- Lines that begin with a "+" are a continuation of the previous grammar rule.

The Palimpsest Grammar for *LOADBAY*

```
*** Below is the CONTROL component of the palimpsest grammar

<loadbay>            ::= <remove_all_fuel> <add_one_food> <load_bays>
<remove_all_fuel>    ::= <remove_fuel> <remove_all_fuel>
<remove_all_fuel>    ::=
<add_one_food>       ::= <add_food>
<add_one_food>       ::=
<load_bays>          ::= <check_load> <load_loop>
<load_loop>          ::= <load_loop> <strategy_a>
<load_loop>          ::= <load_loop> <strategy_b>
<load_loop>          ::=
<strategy_a>         ::= <check_a1> <open> <load_large> <load_med>
+                        <load_small>
<strategy_a>         ::= <check_a2> <open> <load_large> <load_med>
+                        <load_small>
<open>               ::= <open_new_bay>
<open>               ::=
<load_large>         ::= <load_large_item> <load_large>
<load_large>         ::=
<load_med>           ::= <load_med_item> <load_med>
<load_med>           ::=
<load_small>         ::= <load_small_item> <load_small>
<load_small>         ::=
<strategy_b>         ::= <check_b> <strategy_b_loop>
<strategy_b_loop>    ::= <swap_items> <strategy_b_loop>
<strategy_b_loop>    ::= <move_item> <strategy_b_loop>
<strategy_b_loop>    ::=

*** Below is the PRODUCTION MEMORY component of the palimpsest grammar,
*** with associated disambiguation code, and semantics.

<remove_fuel> ::= fuel
    remove(element => fuel);

<add_food> ::= not_food
    add_and_classify(category => item,
                     type     => food,
                     in_bay   => none,
                     size     => 10,
                     name     => genname(item));
```

```
<open_new_bay> ::= anyitem not_openbay closedbay
/   if test_sign = positive then
/     satisfied := closedbay.space >= anyitem.size;
/   else
/     satisfied := not  ((not_openbay <> nil)
/                   and  (not_openbay.space >= anyitem.size));
/   end if;
    change_and_classify(element => closedbay,
                        open    => true);

<load_large_item>  ::= largeitem openbay not_item
/   if test_sign = positive then
/     satisfied := openbay.space >= largeitem.size;
/   else
/     satisfied := not  ((not_item <> nil)
/                   and  (not_item.type = largeitem.type)
/                   and  (not_item.in_bay = openbay.name));
/   end if;
    change_and_classify(element => openbay,
                        space   => openbay.space - largeitem.size);
    change_and_classify(element => largeitem,
                        in_bay  => openbay.name);

<load_med_item> ::= meditem openbay not_item
/   if test_sign = positive then
/     satisfied := openbay.space >= meditem.size;
/   else
/     satisfied := not  ((not_item <> nil)
/                   and  (not_item.type = meditem.type)
/                   and  (not_item.in_bay = openbay.name));
/   end if;
    change_and_classify(element => openbay,
                        space   => openbay.space - meditem.size);
    change_and_classify(element => meditem,
                        in_bay  => openbay.name);

<load_small_item> ::= smallitem openbay not_item
/   if test_sign = positive then
/     satisfied := openbay.space >= smallitem.size;
/   else
/     satisfied := not  ((not_item <> nil)
/                   and  (not_item.type = smallitem.type)
/                   and  (not_item.in_bay = openbay.name));
/   end if;
    change_and_classify(element => openbay,
                        space   => openbay.space - smallitem.size);
    change_and_classify(element => smallitem,
                        in_bay  => openbay.name);
```

```
<swap_items> ::= openbay openbay item item not_item not_item
/   if test_sign = positive then
/     satisfied := (openbay-2.space > openbay-1.space)
/              and (item-1.size < item-2.size)
/              and (item-1.in_bay = openbay-1.name)
/              and (item-2.size ≤ openbay-1.space + item-1.size)
/              and (item-2.in_bay = openbay-2.name);
/   else
/     satisfied := not ((not_item-1 <> nil)
/                  and  (not_item-1.name <> item-2.name)
/                  and  (not_item-1.type = item-2.type)
/                  and  (not_item-1.in_bay = item-2.in_bay))
/              and not ((not_item-2 <> nil)
/                  and  (not_item-2.name <> item-1.name)
/                  and  (not_item-2.type = item-1.type)
/                  and  (not_item-2.in_bay = item-1.in_bay));
/   end if;
    change_and_classify(element => item-2,
                        in_bay  => openbay-1.name);
    change_and_classify(element => item-1,
                        in_bay  => openbay-2.name);
    change_and_classify(element => openbay-2,
                        space   => openbay-2.space - item-2.size
                                   + item-1.size);
    change_and_classify(element => openbay-1,
                        space   => openbay-1.space - item-1.size
                                   + item-2.size);

<move_item>       ::= openbay openbay item not_item
/   if test_sign = positive then
/     satisfied := (openbay-2.space > openbay-1.space)
/              and (item.size ≤ openbay-1.space)
/              and (item.in_bay = openbay-2.name);
/   else
/     satisfied := not ((not_item <> nil)
/                  and  (not_item.type = item.type)
/                  and  (not_item.in_bay = openbay-1.name));
    change_and_classify(element => item,
                        in_bay  => openbay-1.name);
    change_and_classify(element => openbay-2,
                        space   => openbay-2.space - item.size);
    change_and_classify(element => openbay-1,
                        space   => openbay-1.space + item.size);

<check_load>      ::=
    add_and_classify(category => strategy,
                     swapped  => true);

<check_a1>        ::= swapped

<check_a2>        ::= anyitem not_wastedspace
```

```
<check_b>          ::= wastedspace unswapped
    change_and_classify(element => unswapped,
                        swapped => true);
```

B.3. Execution of the *LOADBAY* Palimpsest Parser

A partial execution trace is shown below for the palimpsest parser derived from the *LOADBAY* controlled production system and the working memory of FIGURE B.1. Every state entered by the parser is represented by lines similar to the following:

Action: Shift *largeitem*. Element class *LARGEITEM* contains at least *item1*
Stack: <remove_all_fuel> <add_one_food> <a_ok> openbay largeitem
Effect: None

where the ***Action*** line describes the action just taken by the parser, the ***Stack*** line describes the (topmost) symbols on the parse stack, and the ***Effect*** line describes the effect this action has on working memory.

Action: Initialize palimpsest parser
Stack:
Effect: Working memory is that of FIGURE B.1.

Action: Shift *fuel*. Element class *FUEL* contains at least *item13*
Stack: fuel
Effect: None

Action: Fire production ***remove_fuel***
Stack: <remove_fuel>
Effect: A fuel item (e.g., *item13*) is removed from working memory

---

Action: Shift *fuel*. Element class *FUEL* contains *item2*
Stack: <remove_fuel> fuel
Effect: None

Action: Fire production ***remove_fuel***
Stack: <remove_fuel> <remove_fuel>
Effect: A fuel item (e.g., *item2*) is removed from working memory

---

Action: Reduce by *<remove_all_fuel>* $\rightarrow \varepsilon$
Stack: <remove_fuel> <remove_fuel> <remove_all_fuel>
Effect: None

Action: Reduce by *<remove_all_fuel>* $\rightarrow$ *<remove_fuel>* ...
Stack: <remove_fuel> <remove_all_fuel>
Effect: None

Action: Reduce by *<remove_all_fuel>* $\rightarrow$ *<remove_fuel>* ...
Stack: <remove_all_fuel>
Effect: None

Action: Shift *not_food*

Stack: <remove_all_fuel> not_food

Effect: None

Action: Fire production *add_food*

Stack: <remove_all_fuel> <add_food>

Effect: A food item is added to working memory (e.g., *item15*)

Action: Reduce by *<add_one_food>* → *<add_food>*

Stack: <remove_all_fuel> <add_one_food>

Effect: None

Action: Reduce by *<check_load>* → ε

Stack: <remove_all_fuel> <add_one_food> <check_load>

Effect: Add a working memory element from element class *swapped*

Action: Reduce by *<check_load>* → ε

Stack: <remove_all_fuel> <add_one_food> <check_load>

Effect: Add a working memory element from element class *swapped*

Action: Reduce by *<load_loop>* → ε

Stack: <remove_all_fuel> <add_one_food> <check_load> <load_loop>

Effect: None

Action: Shift *swapped*. Element class *SWAPPED* contains one element

Stack: ... <check_load> swapped

Effect: None

Action: Reduce by *<check_a1>* → *swapped*

Stack: ... <check_load> <check_a1>

Effect: None

Action: Shift *anyitem*. Element class *ANYITEM* contains at least *item1*

Stack: ... <check_load> <check_a1> anyitem

Effect: None

Action: Shift *not_openbay*

Stack: ... <check_load> <check_a1> anyitem not_openbay

Effect: None

Action: Shift *closedbay*. Element class *CLOSEDBAY* contains at least *bay1*

Stack: ... <check_load> <check_a1> anyitem not_openbay closedbay

Effect: None

Action: Fire production *open_new_bay*

Stack: ... <check_load> <check_a1> <open_new_bay>

Effect: The open attribute of *bay1* becomes *true*. *bay1* becomes an *openbay*

Action: Reduce by *<open>* → *<open_new_bay>*

Stack: ... <check_load> <check_a1> <open>

Effect: None

Action: Shift *largeitem*. Element class *LARGEITEM* contains at least *item1*

Stack: ... <check_a1> <open> largeitem

Effect: None

Action: Shift *openbay*. Element class *OPENBAY* contains *bay1*
Stack: ... <check_a1> <open> largeitem openbay
Effect: None

Action: Shift *not_item*
Stack: ... <check_a1> <open> largeitem openbay not_item
Effect: None

Action: Attempt to fire ***load_large_item***. No large item fits. Backtrack and reduce by <load_big> $\rightarrow \varepsilon$
Stack: ... <check_a1> <open> <load_big>
Effect: None

Action: Shift *meditem*. Element class *MEDITEM* contains at least *item7*
Stack: ... <check_a1> <open> <load_big> meditem
Effect: None

Action: Shift *openbay*. Element class *OPENBAY* contains *bay1*
Stack: ... <check_a1> <open> <load_big> meditem openbay
Effect: None

Action: Shift *not_item*
Stack: ... <check_a1> <open> <load_big> meditem openbay not_item
Effect: None

Action: Fire production ***load_med_item***.
Stack: ... <check_a1> <open> <load_big> <load_med_item>
Effect: ***in_bay*** of *item8* is set to *bay1* and ***space*** of *bay1* is reduced by 39

---

and so on...

VITA

Louis Paul Slothouber

Born in Arlington, Virginia, May 8, 1961. Graduated from Osbourn Park Senior High School in Manassas, Virginia, June 1979. B.S., College of William and Mary, 1983. M.S., College of William and Mary, 1987. Ph.D., Computer Science, College of William and Mary, 1989.

Since September, 1988, the author has been employed as an Assistant Professor in the Department of Computer Science, University of Houston.

Research interests include production systems, problem representation, knowledge representation, and natural language processing.