A Logic-Based Grammar Formalism Incorporating Feature-Structures and Inheritance

Harry H. Porter, III
Sc.B., Brown University, 1978

A dissertation submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree

**Doctor of Philosophy
in Computer Science**

December, 1988

The dissertation "A Logic-Based Grammar Formalism Incorporating Feature-Structures and Inheritance" by Harry H. Porter, III has been examined and approved by the following Examination Committee:

David Maier
Professor, Thesis Advisor

Richard B. Kieburtz
Professor and Chairman

Dan W. Hammerstrom
Associate Professor and
Adaptive Systems, Inc.

Brian Philips
Intelligent Business Systems, Inc.

For my family

Table of Contents

Abstract		1
Introduction		2
Part 1: Background and Context		5
1.	**Logic-Based Grammar Formalisms**	6
1.1.	Introduction	6
1.2.	First-Order Logic and Logic Programming	6
1.3.	Context-Free Grammars	10
1.4.	Definite Clause Grammars	11
1.4.1.	DCG Notation	15
1.4.2.	Beyond Context-Free Rules: Additional Arguments	16
1.4.2.1.	Other Uses for Arguments	17
1.5.	Logic-Based Approaches to NL Processing	19
2.	**Unification-Based Grammar Formalisms**	23
2.1.	Introduction	23
2.2.	Lexical-Functional Grammar	23
2.2.1.	Functional Unification Grammars	34
2.3.	PATR-II	35
Part 2: Inheritance Grammar		43
3.	**Introduction to Inheritance Grammar**	44
3.1.	Introduction	44
3.2.	The IS-A Relation Among Feature Values	45
3.3.	Viewing ψ-terms as Enhanced Feature Structures	46
3.4.	Unification of ψ-terms	50
3.5.	Relationship to F-structures and First-Order Terms	53
3.6.	Inheritance Grammars	55
3.7.	Search Strategy	56
3.8.	Type-Class Reasoning in Parsing	58
3.9.	Other Grammatical Applications of Taxonomic Reasoning	59
3.10.	Implementation	60
4.	**Definition of Inheritance Grammar**	61
4.1.	Introduction	61
4.2.	ψ-Terms	61

4.3. A Lattice of ψ-Terms 80
4.4. Extending Horn Clause Logic to ψ-Logic 95
4.5. A Resolution Rule for ψ-Clauses 103
4.6. A Model-Theoretic Semantics for ψ-Programs 111
4.6.1. Soundness and Completeness 121
4.6.2. Soundness of ψ-Resolution 123

Part 3: Implementation 133

5. Implementations of Inheritance Grammar 134

5.1. Introduction 134
5.2. The Interpreter System 135
5.3. The Compiler System 141
5.4. The Grammar Development Environment 152
5.5. Implementation-Specific Grammar Details 164
5.6. Built-In Predicates 170

6. Additional Evaluation Strategies 172

6.1. Introduction 172
6.2. Earley Deduction 174
6.2.1. Description of Earley Deduction 174
6.2.1.1 Discussion 179
6.2.2. Application of Earley Deduction to ψ-Logic 180
6.2.3. Soundness of Earley Deduction 181
6.2.4. Completeness of Earley Deduction 183
6.2.5. Termination for Datalog 190
6.2.6. Implementation 192
6.2.6.1. Indexing the Clauses 193
6.2.6.2. Optimizations for Datalog Programs 195
6.2.6.3. Experimental Evaluation of the Optimizations 204
6.3. Extension Tables 211
6.4. Staged Depth-First Search Strategy 219
6.5. Other Evaluation Strategies 220
6.6. Discussion of Evaluation Strategies 221

Conclusions 223

References 225

Appendices 235

A.1. An Example Grammar 235
A.2. Implementation Size Statistics 238
A.3. Comparison of PATR-II and IG by Example 240
A.4. An Extended Inheritance Grammar 250

Biographical Note 275

List of Figures

1.1 Syntax Of Predicate Logic
2.1 An F-Structure
2.2 An F-Structure Containing Shared Attributes
2.3 A Simple Lexical-Functional Grammar
2.4 A Context-Free Parse
2.5 The Instantiated Equations
2.6 The Solutions
2.7 A Reentrant F-Structure in PATR-II Notation
2.8 The DAG representation of an F-Structure.
3.1 An IS-A Ordering with Multiple Inheritance
3.2 The Corresponding Signature Lattice
3.3 Graphical Representation of a Cyclic ψ-Term
3.4 An F-Structure
4.1 The address domain A_2
4.2 The address domain A_2'
4.3 Infinite Representation of A_1
4.4 A Representation of t_1
4.5 Graphical representation of t_2
4.6 Term t_2 With Some Variables Elided
4.7 Graphical representation of t_3
4.8 The join of two terms
4.9 Graphical representation of C_1
4.10 An Interpretation
5.1 The Representation of Two ψ-Terms
5.2 The Result of their Unification
5.3 The Representation of a ψ-Clause
5.4 Using the Compiler
5.5 Bringing Up The IG System
5.6 A Rule Browser
5.7 A Dialog Window
5.8 Parsing a Statement
5.9 Viewing an Asserted Clause
5.10 Parsing a Question
5.11 Editing a Signature
5.12 Finding a GLB
5.13 Syntax Accepted by the Implementation
6.1 An Infinite Deduction Sequence
6.2 A Completed Earley Deduction
6.3 Earley Deduction using ψ-Logic
6.4 An Earley Deduction Tree
6.5 A Proof Tree
6.6 An Interior Node

6.7 Resulting Earley Deduction Tree
6.8 A Constructed Earley Deduction Tree
6.9 A Complete-Key Index
6.10 A Selected-Literal-Key Index
6.11 A Program-Rule-Head Index
6.12 The Representation of a Datalog Program
6.13 A Test Program
6.14 The Execution Times for Figure 6.13
6.15 Another Test Program
6.16 The Execution Times for Figure 6.15
6.17 Analysis of Subsumption Checking

Abstract

Incorporating Inheritance and Feature Structures
into a Logic Grammar Formalism

Harry H. Porter, III, Ph.D.
Oregon Graduate Center, 1988

Supervising Professor: David Maier

The use of Definite Clause Grammars (DCGs) to describe Natural Language grammars as logic programs has been particularly fruitful in building NL interfaces for relational databases. First-order logic, however, suffers from several shortcomings as a language for knowledge representation and computational linguistics.

This dissertation describes a new logic, called ψ-logic, remedying these shortcomings. NL processing is one application of ψ-logic and the *Inheritance Grammar* formalism, whose semantics are rooted in ψ-logic, is also defined. First-order logic is a special case of ψ-logic and, consequently, every DCG is an Inheritance Grammar. This logic differs from traditional logic in that the primary data structure encompasses the feature-based structures used in unification-based grammatical formalisms such as PATR. In addition, an ordering on the symbols of the grammar facilitates taxonomic reasoning, a problematic task in DCGs.

An interpreter for ψ-logic programs has been implemented (in Smalltalk) and various implementation techniques have been explored. In addition, several Inheritance Grammars have been implemented and the perspicuity of the Inheritance Grammar formalism is discussed.

Introduction

The ability to use symbols to communicate and process information is the characteristic that distinguishes human intelligence from animal behavior. The goal of processing Natural Language with a computer is motivated by both practical concern – to learn how it *can be* done – and theoretical curiosity – to learn how it *is* done. The former aim motivates this dissertation, giving it a more computational flavor than most linguistic inquiry. To summarize, this work introduces a formal language for expressing Natural Language grammars, called the *Inheritance Grammar* formalism. We will provide a formal semantics, describe our implementation, and show example applications to NL processing problems.

It is important to locate new research in the context of existing work and there are two general approaches that can be used to introduce a new formalism. First, the new formalism can be described, followed by a description of other similar formalisms under the rubric of *related work*. Alternatively, the existing work can be presented first and followed by the formalism being introduced. The first approach seems more suited for situations where there is a loose coupling between the old and the new. The second is preferred when there is a stronger relationship since the preliminary review serves to focus attention on the aspects of existing work that should be compared. When the new formalism is introduced, its similarities and dissimilarities, strengths and weaknesses are then easily exposed.

We will adopt the second strategy, setting the stage in Part One (Chapters 1-2) with a review of similar, existing research. This work is divided into two categories:

logic-based grammatical formalisms and *unification-based* grammatical formalisms[1]. Chapter 1 begins with a concise refresher on logic programming, which can be safely skipped by the reader familiar with Prolog and its theoretical underpinnings. We then review the Definite Clause Grammar formalism and discuss its application in computational linguistics research. Chapter 2 discusses two unification-based formalisms, called Lexical-Functional Grammar and PATR-II, which at first sight may appear different from the logic-based formalisms but provide important context for Inheritance Grammar.

In Part Two, we introduce (Chapter 3) and formally describe (Chapter 4) Inheritance Grammar and its underlying logic. In Chapter 3, we view Inheritance Grammar as a generalization of both Definite Clause Grammar and PATR-II. (In fact, every Definite Clause Grammar is a legal Inheritance Grammar and there is a simple translation from PATR-II into Inheritance Grammar.) In Chapter 4, we extend the traditional first-order logic system to the *ψ-logic* system, giving several definitions and results analogous to those in the classical development of first-order logic. The semantics of the Inheritance Grammar formalism is based on this new logic. From this point of view, NL processing is just one application of ψ-logic, the one that motivated its existence.

We conclude in Part Three by discussing execution strategies for implementing ψ-logic and Inheritance Grammars. Using the Smalltalk environment, we constructed two concrete implementations, both based on Prolog's fast, depth-first execution strategy. First, we implemented an interpreter for ψ-logic and then, to achieve greater speed, we implemented a compiler and run-time execution system. These implementations are

[1] The term *unification-based* is used by linguists to circumscribe work based on an operation similar to, but distinct from, the unification of first-order logic.

described in Chapter 5, while three example grammars we executed are listed in the appendices. The presentation of the formal semantics in Chapter 4 concludes by describing what an implementation must do to be *complete*. In Chapter 6 several complete evaluation strategies are discussed. We attempt to evaluate these strategies and include information gained from several experiments we performed on a strategy called *Earley Deduction*.

I owe thanks to many people for providing emotional and intellectual support throughout this research. The most important is of course my wife, Nancy McCarter, a wonderful individual whom I love very deeply. Thank you, Nancy. My advisor and friend, David Maier, deserves credit for all of the good ideas presented herein. By bringing his broad experience and deep thoughtfulness to bear on the minutiae of ψ-logic, he forced my brain to dangerously exceed its rated capacity on a number of occasions. Thank you, Dave. The other members of my committee—Dick Kieburtz, Dan Hammerstrom, and Brian Phillips—have shared many exciting ideas with me. Thank you. I am also grateful to a number of other people who have transmitted information-intense input to this process. This list certainly includes Hassan Ait-Kaci, Lauri Karttunen, Martin Kay, Fernando Pereira, and David S. Warren. Finally, through the research community at the Oregon Graduate Center and in Portland generally, I have come into contact with a number of really bright and entertaining computer-nerds. Thank you all for existing; life would be a lot less interesting without you.

Part 1
Background and Context

Chapter 1: Logic-Based Grammar Formalisms

1.1. Introduction

This chapter surveys the logic-based approach to grammatical analysis. After reviewing the notation of first-order predicate logic and Logic Programming (see, for example, [Maier and Warren 1988], [Apt and Van Emden 1982] or [Van Emden and Kowalski 1976] for a review of Logic Programming), this chapter describes the Definite Clause Grammar (DCG) formalism. In Chapters 3 and 4, we introduce a new grammar formalism; the goal of this chapter and the next is to provide the background and context necessary to evaluate the new formalism vis-a-vis existing grammatical research.

1.2. First-Order Logic and Logic Programming

We will use the syntax of first-order predicate logic given in Figure 1.1. In the extended BNF (eBNF) we will use throughout this document, brackets enclose optional constituents, the vertical bar separates alternate constituents, and ellipses are used (loosely) to indicate repetition in lists of constituents and separators. Other punctuation symbols – including parentheses – are terminals and are in boldface when possible. Non-terminals defined by the grammar are capitalized and non-terminals defined elsewhere are in lower case.

We write $A \models E$ to mean that axioms A logically imply expression E vis-a-vis model theoretic semantics and we write $A \vdash E$ to mean that a specific proof procedure proves E. A proof procedure is called *sound* (also *consistent*) if

```
TERM          ::=  constant
                   | function TERMLIST
                   | variable
TERMLIST      ::=  ( TERM , TERM , ... TERM )
ATOM          ::=  predicate [ TERMLIST ]
EXPRESSION    ::=  ATOM
                   | true
                   | false
                   | ¬ EXPRESSION
                   | EXPRESSION BINARYOP EXPRESSION
                   | ∃ variable . EXPRESSION
                   | ∀ variable . EXPRESSION
                   | ( EXPRESSION )
BINARYOP      ::=  ∧ | ∨ | ⇒ | ⇔
```

Figure 1.1: Syntax Of Predicate Logic

$$A \vdash E \text{ implies } A \models E$$

and *complete* if

$$A \models E \text{ implies } A \vdash E$$

Clauses are disjunctions of literals where all variables are universally quantified. Horn clauses have at most one positive literal and zero or more negative literals. Definite clauses have exactly one positive literal (the head literal) and zero or more negative literals (the body). While all logic expressions can be transformed into a set (i.e., a conjunction) of clauses, not all expressions can be transformed into a set of Horn clauses. The premise of logic programming is that many interesting reasoning problems can be expressed using Horn clause sets.

A *Logic Program* consists of zero or more definite clauses (called the database rules or axioms) and a single clause with no positive literals (the query), which is the theorem to be proved. The standard convention is to write a definite clause

$$p_0 \vee \neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k$$

as

$$p_0 \text{ :- } p_1, p_2, \cdots, p_k.$$

and to write a query

$$\neg p_1 \vee \neg p_2 \vee \ldots \vee \neg p_k$$

as

$$\text{:- } p_1, p_2, \cdots, p_k.$$

To understand the logical interpretation of a rule and the quantification of the variables within the rule, let $\overline{Y}$ be the variables that occur in the body but not in the head of the rule and let $\overline{X}$ be all remaining variables. The following expressions are seen to be equivalent by standard identities of first-order logic.

$$\forall \overline{X}.\ \forall \overline{Y}.\ (p_0(\overline{X}) \vee \neg p_1(\overline{X},\overline{Y}) \vee \ldots \vee \neg p_k(\overline{X},\overline{Y}))$$

$$\forall \overline{X}.\ (p_0(\overline{X}) \vee \forall \overline{Y}.\ (\neg p_1(\overline{X},\overline{Y}) \vee \ldots \vee \neg p_k(\overline{X},\overline{Y})))$$

$$\forall \overline{X}.\ (p_0(\overline{X}) \vee \forall \overline{Y}.\ \neg(p_1(\overline{X},\overline{Y}) \wedge \ldots \wedge p_k(\overline{X},\overline{Y})))$$

$$\forall \overline{X}.\ (p_0(\overline{X}) \vee \neg \exists \overline{Y}.\ (p_1(\overline{X},\overline{Y}) \wedge \ldots \wedge p_k(\overline{X},\overline{Y})))$$

$$\forall \overline{X}.\ (\exists \overline{Y}.\ (p_1(\overline{X},\overline{Y}) \wedge \ldots \wedge p_k(\overline{X},\overline{Y})) \Rightarrow p_0(\overline{X}))$$

In words, the rule is equivalent to the following (declarative) statement for any X: "If $p_1(X,Y)$ and ... and $p_k(X,Y)$ are true (for at least one Y) then $p_0(X)$ is true." Expressing this relationship procedurally: "To show that $p_0(X)$ is true (for any X), it is sufficient to find a Y such that $p_1(X,Y)$ and ... and $p_k(X,Y)$ are true."

A query asks "Does there exist an X such that $p_1(X)$ and ... and $p_k(X)$ are true." Letting $\overline{X}$ represent the variables in the query, this question is:

$$\exists \overline{X}.\ (p_1(\overline{X}) \wedge \ldots \wedge p_k(\overline{X}))$$

For refutation proofs, the query is first negated and then converted to clause form:

$$\neg(\exists \overline{X}.\ (p_1(\overline{X}) \wedge \ldots \wedge p_k(\overline{X})))$$

$$\forall \overline{X}.\ \neg(p_1(\overline{X}) \wedge \ldots \wedge p_k(\overline{X}))$$

$$\forall \overline{X}.\ (\neg p_1(\overline{X}) \vee \ldots \vee \neg p_k(\overline{X}))$$

Logic programming consists of using a resolution-based theorem prover to prove a query. The resolution rule is summarized as follows: If a rule head matches a predicate in the query body, then we can find a solution for the query if we can find solutions for all of the predicates in the body of the rule and we can find solutions for all the remaining goals in the query. The computational interpretation parallels this summary: A program task consists of a conjunction of procedures to be called. The rules are used to execute the calls. If a rule head unifies with a call in the task body, then we can execute the task by executing all the calls in the rule body and executing all the remaining calls in the task.

The Prolog language imposes an order on the set of clauses and an order on the literals within the clauses. In choosing a literal from a goal or rule body to match against the rules, Prolog tries to solve the literals in the order listed. In choosing a rule to match against a selected literal, Prolog tries the rules in order. In searching for a refutation proof, all the literals in the body must be solved because they are conjoined (motivating and-parallelism in multiprocess implementations) while only one rule that solves a selected literal needs to be found (motivating or-parallelism). When a particular rule fails to solve a goal literal, Prolog backtracks to try successive rules in the database; when a literal with a goal or rule body can not be solved, the entire goal or rule fails.

Finding a refutation can be viewed as a problem of searching an and-or tree. The Prolog execution rule is a depth-first search. Since the tree may have a finite solution among many infinite branches, a depth-first search may follow an infinite path and fail to find an existing solution. Prolog's execution strategy is not complete: there may exist solutions to a given logic program that Prolog will never find. Next, we show how context-free parsing can be done with Logic Programs by turning the search for a proof tree into a search for parse tree.

1.3. Context-Free Grammars

Recall that a context-free grammar (CFG) describes a context-free language (CFL), also called a Chomsky type-2 language. The grammar identifies the set of sentences (input strings) in the language. There are many well studied context-free parsing algorithms to analyze a potential sentence and determine whether it is syntactically legal [e.g., Aho and Ullman 1972]. A parser usually produces a parse tree, which linguists often call a *phrase structure tree*, a *constituent structure tree*, or a *syntax tree* to distinguish it from other deeper representations of the sentence.

Context-free grammars are represented naturally in Backus-Naur Form (BNF), the rules of which contain a non-terminal on the left side and a sequence of terminals and non-terminals on the right. In the parse tree constructed, the internal nodes are labeled with non-terminals and the leaves are labeled with terminals. Lexical (or morphological) analysis categorizes the words of an input into lexical categories (word classes), which appear as terminals in the grammar.

In spite of persisting debate to the contrary [e.g., Pullum 1984], it is clear that context-free languages are by themselves inadequate to describe natural languages. Nevertheless, much of the superficial structure of language can be described quite easily

with CF rules. Almost every NL system has a syntactic component founded on a CFG skeleton and augmented with constraints and code to construct some form of semantic representation. Constraints (for example, the *number agreement* constraint between determiner and head noun or the constraint between noun phrase and verb sequence) serve to restrict the applicability of CF rules. Definite Clause Grammars make it easy to integrate the context-free analysis with such constraint checking and structure building.

1.4. Definite Clause Grammars

Colmerauer and Kowalski observed that there is a simple translation from context-free BNF rules to definite clauses such that the problem of parsing a given sentence is equivalent to proving a theorem [Colmerauer 1978, Kowalski 1979]. Furthermore, when translated into logic, there is an obvious extension to the context-free skeleton that makes such tasks as constraint checking and tree building easy. The basic formalism used to express NL grammars in logic is called *Definite Clause Grammars* [Pereira and Warren 1980].

Consider the following BNF rule:

$$S ::= S_1 \;\; S_2 \;\; \ldots \;\; S_k$$

where the S_i are non-terminals. It can be translated into the following logic clause:

$$s(P_1,\ P_{k+1})\ \text{:-}\ \ s_1(P_1,\ P_2),\ s_2(P_2,\ P_3),\ \ldots\ ,\ s_k(P_k,\ P_{k+1}).$$

During execution, the variables P_i will be instantiated to positions in the input. By convention, positions fall between the word symbols and can be identified by integers as in the input

$_0$ **The** $_1$ **messenger** $_2$ **delivered** $_3$ **the** $_4$ **news** $_5$

Declaratively, this rule can be read as "There is an S constituent in the input between positions P_1 and P_{k+1} if there is an S_1 constituent between position P_1 and P_2 and ... and an S_k constituent between position P_k and P_{k+1}."

If the right side of the BNF rule contains a terminal as, for example, the word w in

$$S ::= \ldots, S_{i-1}, \mathbf{w}, S_{i+1}, \ldots$$

it is translated into the following clause:

$$\mathtt{s}(P_1, P_{k+1}) \text{ :- } \ldots, \mathtt{s}_{i-1}(P_{i-1}, P_i), \mathtt{connects}(\mathbf{w}, P_i, P_{i+1}), \mathtt{s}_{i+1}(P_{i-1}, P_{i+2}), \ldots$$

This clause can be read as "There is an S constituent in the input between P_1 and P_{k+1} if there is ... and an S_{i-1} between P_{i-1} and P_i and a word **w** between positions P_i and P_{i+1} and an S_{i+1} between P_{i+1} and P_{i+2} and ... ." For example, the grammar

S ::= NP VERB NP
NP ::= **the** NOUN
NOUN ::= **messenger**
NOUN ::= **news**
VERB ::= **delivered**

is represented in definite clauses as

```
s(P1, P4) :- np(P1, P2), verb(P2, P3), np(P3, P4).
np(P1, P3) :- connects(the, P1, P2), noun(P2, P3).
noun(P1, P2) :- connects(messenger, P1, P2).
noun(P1, P2) :- connects(news, P1, P2).
verb(P1, P2) :- connects(delivered, P1, P2).
```

The input sentence to be parsed is then specified with a series of *connects* facts, such as:

```
connects(the, 0, 1).
connects(messenger, 1, 2).
connects(delivered, 2, 3).
connects(the, 3, 4).
connects(news, 4, 5).
```

To use a Definite Clause Grammar as a parser, the grammar is executed as a logic program. To determine whether the input is syntactically correct, i.e. whether a legal constituent S lies between positions 1 and 5, the following goal is evaluated:

```
:- s(1, 5).
```

The depth-first evaluation strategy of Prolog applied to a DCG performs a top-down parse, attempting to prove the input is syntactically correct. Different execution strategies (e.g., Earley Deduction), resulting in correspondingly different parse strategies, will be discussed later. The deductive mechanism of the DCG formalism is a direct consequence of the execution strategy used for the corresponding logic program. The incompleteness of Prolog deduction in implementing logic program semantics motivates the discussion of different, complete strategies in Chapter 6.

Above, input positions were represented with integers. An optimization is to represent positions with lists[1] of words instead of integers. In particular, a position P in the input is represented as the list of all words following position P. The `connects` facts then become

[1] A list `[X1, X2, ... , Xk]` is syntactic shorthand for the first-order term `cons(X1, cons(X2, ... cons(Xk, nil) ... ))`. The notation `[X1, X2, ... , Xk | Y]` is short for `cons(X1, cons(X2, ... cons(Xk, Y) ... ))` and `[]` is short for `nil`. Most Prolog implementations accept list notation and we shall use it throughout.

```
connects(the, [the,messenger,delivered,the,news],
              [messenger,delivered,the,news]).
connects(messenger, [messenger,delivered,the,news],
                    [delivered,the,news]).
connects(delivered, [delivered,the,news],
                    [the,news]).
connects(the, [the,news], [news]).
connects(news, [news], []).
```

All such rules can be collapsed into one general purpose rule:

```
connects(W, [W | List], List).
```

thus making the program independent of particular input strings. The parsing can then be initiated with a query such as

```
:- s([the,messenger,delivered,the,news], []).
```

When the terminals in the CF grammar represent word classes (noun, adjective, etc.) a lexicon, which maps specific words into their classes, can be represented as a sequence of rules such as:

```
noun(P1, P2) :- connects(man, P1, P2).
noun(P1, P2) :- connects(woman, P1, P2).
adjective(P1, P2) :- connects(big, P1, P2).
adjective(P1, P2) :- connects(red, P1, P2).
    etc.
```

This organization for the lexicon is not particularly efficient under Prolog execution since every `noun` rule must be tried when the grammar expects a noun, making the speed of parsing proportional to the size of the lexicon. A more efficient technique is to use the indexing mechanisms built in to some Prolog implementations. For example, the same lexicon would be coded as:

```
noun(P1, P2) :- connects(W, P1, P2), isNoun(W).
isNoun(man).
isNoun(woman).
    etc.

adjective(P1, P2) :- connects(W, P1, P2), isAdjective(W).
isAdjective(big).
isAdjective(red).
    etc.
```

Another improvement is to preprocess out explicit calls to `connects`. For example, the following rule

```
noun(P1, P2) :- connects(W, P1, P2), isNoun(W).
```

can be rewritten as

```
noun([W | P2], P2) :- isNoun(W).
```

Likewise, the rule

```
s(P1, P4) :- s1(P1, P2), connects(W, P2, P3), s3(P3, P4).
```

can be condensed to

```
s(P1, P4) :- s1(P1, [W | P3]), s3(P3, P4).
```

The `connects` rule can then be eliminated altogether.

1.4.1. DCG Notation

DCGs are really nothing more than stylized logic programs with conventions for the use of position variables to encode the CFG. The position variables are used so regularly, in fact, that a simple bit of syntactic sugar allows them to be generated automatically, leaving the essence of the grammar unfettered with a proliferation of position variables. The standard is to write a BNF rule as:

$$s \text{ --> } s_1,\ s_2,\ \ldots,\ s_k.$$

When the right side includes a terminal, it is inclosed in brackets:

```
s --> ... , s_{i-1}, [w], s_{i+1}, ...
```

A simple syntactic transformation recognizes the --> symbol and translates the rule into a legal clause by adding the appropriate position variables. A rule such as

RELCLAUSE ::= ϵ

is coded as

```
relclause --> .
```

When the grammar writer intends a clause such as:

```
noun (P1,P2) :- connects (W,P1,P2), isNoun(W).
```

which includes a goal that is not to have position variables, the DCG notation allows the goal to be explicitly escaped with braces. So, the clause above can be expressed in DCG notation as:

```
noun --> [W], { isNoun(W) }.
```

1.4.2. Beyond Context-Free Rules: Additional Arguments

The obvious extension to the context-free Definite Clause notation allows non-terminal symbols to have additional arguments. For example, the rule:

```
s --> np(Num), vp(Num).
```

shows how additional arguments might be used to check the *number agreement* constraint between subject and verb phrase. Assume evaluation (parsing) of the np goal results in binding the Num variable to singular or plural. This value is then passed to the vp goal, constraining the form of verb phrase to be accepted.

A familiar benefit of using unification for all data manipulation is that information is used if it is available, but not required to be present. In the sentence *The sheep hit the ball*, the number information is not provided by the subject so the verb phrase is not constrained. The verb phrase is free to add in (unify) either singular (*...hits the ball*) or plural (*...hit the ball*). In the sentence *The sheep wanted the ball*, which is ambiguous since the number is not specified by either subject or verb, Num is never bound.

Contrast Prolog's treatment of arguments with *synthesized attributes* (in the context of parsing), which are always bound in the subconstituent and passed upward to the calling rules, and with *inherited attributes* which are always bound before being passed down to the called constituent [Aho and Ullman 1972, Aho and Ullman 1977, Knuth 1968].

1.4.2.1. Other Uses for Arguments

These ancillary arguments can also be used to build structure. In the next example, we want to construct and return a term representing the parse tree. The second arguments of np and vp are assumed to return structures representing the noun phrase and verb phrase constituents, respectively. The s rule combines them to return the structure for the entire sentence.

```
s(sent(NPtree, VPtree)) --> np(Num,NPtree),
                            vp(Num,VPtree).
```

The use of arguments to build and return parse trees is very regular and additional arguments could easily be generated automatically like the position variables are in DCG notation. In fact, this feature exists in some variants of DCG (e.g., Restriction Grammars and Definite Clause Translation Grammars, to be discussed below). "Standard" DCGs do not have this extension, since the grammar writer often finds it con-

venient to build a more abstract representation, which may differ from a parse tree by (1) rearrangement of nodes and simplification, (2) the inclusion of constraints and features (e.g., number), and (3) the inclusion of computed information such as semantic translations.

In the *compositional* approach to understanding, syntactic constituents – at least the major constituents – return computed translations instead of returning parse trees. In extreme versions of the compositional approach, the meaning of an entire subconstituent (e.g. a noun phrase) is determined in isolation from its context, the enclosing constituent. This approach doesn't scale well; the meaning of a constituent often depends crucially on its context and can not be built in isolation but must be built using inherited information.

Finally, additional arguments can pass semantic fragments (i.e., partial translations), which are combined in arbitrary computation. Typically, semantic fragments are passed both up, down, and sideways, and predicates escaped with braces are used to perform semantic constraint checking and structure building.

To summarize, the context-free skeleton of DCGs is augmented with (1) ancillary arguments to perform constraint checking, (2) ancillary arguments to build and manipulate arbitrary structures, and (3) embedding of arbitrary computations using the brace-escape mechanism. By allowing arbitrary computations, the formalism inherits Chomsky type-0 power from the Turing Machine power of logic programming while still retaining the context-free skeleton in the BNF-like rules. Pereira and Warren [1980] compare the DCG formalism to Augmented Transition Networks [Woods 1970, Bates 1978] and conclude that DCGs are superior.

1.5. Logic-Based Approaches to NL Processing

Research in applying the DCG formalism to Natural Language processing can be divided into two general categories: (1) implementations of NL grammars using DCG techniques and (2) modifications and improvements to the DCG formalism itself.

Early work in DCG grammars by Veronica Dahl argued for the use of logic for both the deductive component of a database and for a natural language interface to that database [Dahl 1979, 1981, 1982; see also Gallaire, et al. 1984]. Logic variables are untyped, while attributes in the relational data model are (typically) typed [Codd 1970]. Dahl pointed out that typing information is needed in the NL component to sort out ambiguities (e.g., how to attach relative clauses). She also discussed semantic data modeling issues in the context of logic-based grammars.

Alain Colmerauer addressed the need to have set-valued expressions and multiple truth values in any logic into which a NL query might be translated. In his more rigorous approach [Colmerauer 1982], he provided a formal semantics for a 3-valued logic system with particular emphasis on translating NL quantifiers (e.g., *every*, *the*, *each*). He also circumscribed a subset of Natural Language (initially French but also English) and specified a simple set of rules for translating sentences and questions into expressions in his logic.

Perhaps the most influential DCG was one written by Michael McCord synthesizing a number of important DCG techniques [McCord 1980, 1982]. Each input sentence is understood in isolation and processed against a relational database over the domains of courses, classes, students, and faculty. The system answers interrogatives and verifies declaratives as correct or incorrect. The linguistic coverage – at least for isolated queries – is quite complete; the primary limitation is the semantic data model. The

grammar's main strengths are its treatment of quantification and its resolution of verb-complement coordination using a semantic hierarchy. McCord's initial DCG was followed by research into the nature of focalizers [McCord 1986] and with the construction of a single-pass semantic backend called SEM [McCord 1984, 1985].

A number of rules in McCord's grammar contain several extra arguments used solely for passing information from one place in a sentence to another distant location. The linguistic phenomenon called *left extraposition* or *fronting* is responsible for this effect. Left extraposition occurs when some linguistic construct Y contains a missing subconstituent X and another, related phrase X′ has been inserted to the left of construct Y. For example, the sentence

The revolution that we believed in was lost.

contains a relative clause, *that we believed in*, which is analyzed as a sentence with a missing element, *the revolution*, which has been moved to the left to become a relative pronoun, *that*. This phenomenon also occurs in auxiliary fronting and wh-questions; in the question

Which critic did the woman with the golden tongue seduce?

the noun phrase *which critic* and the verb auxiliary *did* are both left extraposed.

To handle extraposed constituents, Fernando Pereira introduced *Extraposition Grammars* (XGs) as an extension to the DCG formalism [Pereira 1981]. To execute an XG, a simple translation scheme is used to transform the XG into an efficient definite clause program, which can then be executed by the Prolog interpreter. An Extraposition Grammar was used in the interface to CHAT-80, a logic database of geographic data that used a clever query planning algorithm [Warren 1981, Warren and Pereira 1982]. See also the related work by Dahl and Abramson on *Gapping Grammars* [Dahl

and Abramson 1984].

In analyzing a natural language sentence, several distinct activities are commonly distinguished: context-free parsing, parse tree construction, constraint enforcement, and construction of the semantic representations of syntactic constituents. In DCGs, these tasks are not described or performed separately; instead all are intermixed in the grammar. In another extension to the DCG formalism, called *Definite Clause Translation Grammars* (DCTGs), the CF component of the grammar and the parse tree construction are separated from the other tasks, collectively called semantics [Abramson 1984a, 1984b]. Each DCTG rule then consists of two parts: a syntax part and a semantics part.

Another modification to the DCG formalism was motivated by the Linguistic String Project (LSP), which culminated in a grammar combining BNF rules with *restrictions* [Sager 1981]. Their implementation (in Fortran!) builds a parse tree and checks the restrictions. Hirschman and Puder have adapted this approach to the DCG framework and call the resulting formalism *Restriction Grammars* (RGs) [Hirschman and Puder 1986]. During execution, a parse tree is constructed automatically. A restriction consists of instructions to move a pointer around this parse tree and to check node labels and attributes to validate the parse.

In the work presented here, we take first-order logic (rather than DCGs) as our departure point. We first describe a generalization of first-order logic called *ψ-logic*. Then we extend the DCG notation from first-order logic to ψ-logic to yield the *Inheritance Grammar* (IG) notation for expressing Natural Language grammars.

Before introducing ψ-logic and Inheritance Grammars however, we will survey a different line of research into a family of grammar formalisms called *Unification-Based*

Grammars. While the DCG formalism and its variants were introduced primarily by computer scientists for linguistic processing, unification-based grammars were championed by computational linguists for using computers. One goal of Inheritance Grammars is to gracefully incorporate the important features of the unification-based formalisms into the framework of logic programming.

Chapter 2: Unification-Based Grammar Formalisms

2.1. Introduction

In this chapter we discuss two grammar formalisms based on unification of data structures, but developed independently of the logic programming paradigm. The first, Lexical-Functional Grammar (LFG), was developed primarily as a theory about human language competence and less as a notation for expressing computer-executable grammars. The second, PATR-II, was developed as a general-purpose language in which several types of grammatical theories (e.g., LFG and Functional Unification Grammar) could be expressed in computer-executable form. As such, it can be viewed as a synthesis of other unification-based formalisms.

2.2. Lexical-Functional Grammar

A Lexical-Functional Grammar [Bresnan and Kaplan 1982] is a context-free grammar in which each rule is augmented with one or more equation schemata that express the context-sensitive constraints among the constituents. LFG parsing proceeds in three steps. Given an input sentence to analyze, the first step is to find a context-free parse (called the *c-structure* in LFG terminology). The second step is to examine the rules used in the parse and their associated equation schemata and to instantiate those schemata, thereby producing a set of equations (called the *f-description*) to be solved. The final step is to solve the equations.

The context-free rules describe a superset of the target language and the equations serve to constrain the number of potential parses. When a consistent solution for the equations can be found, the analysis is complete and information about the analysis of the input is captured in that solution. When the equations do not have a consistent solution, another context-free parse is sought and new equations are produced.

The variables in these equations range over *f-structures* (short for *functional structures*). An f-structure is a finite function mapping *attribute labels* (atomic symbols) into *attribute values* (again atoms) or other f-structures. Attributes are also called *features* or *functions*. Semantic formulas or sets (of symbols or f-structures) also appear occasionally as attribute values. F-structures are traditionally shown using a matrix-like notation. In the f-structure shown in Figure 2.1 there are two attribute labels, *SUBJ* and *VP*[1]. The value of each is itself a nested f-structure.

It is important to distinguish between values that are *equal* and values that are *identical* (equality of value versus equality of reference). All identical values are equal, but equal values are not necessarily identical. In Figure 2.1, the *NUM* attribute of the

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{DET} & \text{THE} \\ \text{NUM} & \text{PLURAL} \\ \text{NOUN} & \text{GIRLS} \end{bmatrix} \\ \text{VP} & \begin{bmatrix} \text{NUM} & \text{PLURAL} \\ \text{TENSE} & \text{PRESENT} \\ \text{VERB} & \text{RUN} \end{bmatrix} \end{bmatrix}$$

Figure 2.1: An F-Structure

[1] Many of our LFG examples are from Bresnan and Kaplan [1982] and Winograd [1983].

SUBJ attribute has a value equal to the *NUM* attribute of the *VP* attribute. When two attributes share the same value, they are said to be identical Identical attributes are generally indicated graphically by linking them with a line as in Figure 2.2, but there are also other notations to indicate identity.

In the original description of LFG, it is unclear whether or not cyclic f-structures are allowed A cyclic f-structure is one in which the value of one attribute contains the f-structure itself. A rigorous definition of cyclicity appears in Chapter 4.

Figure 2.3 shows a simple LFG (not necessarily the one that generated the f-structures in Figure 2.1 or Figure 2.2). It contains 3 grammar rules followed by a lexicon of six words. The equation schemata associated with each grammar rule are statements of identity between the different parts of the functional structures to be built and

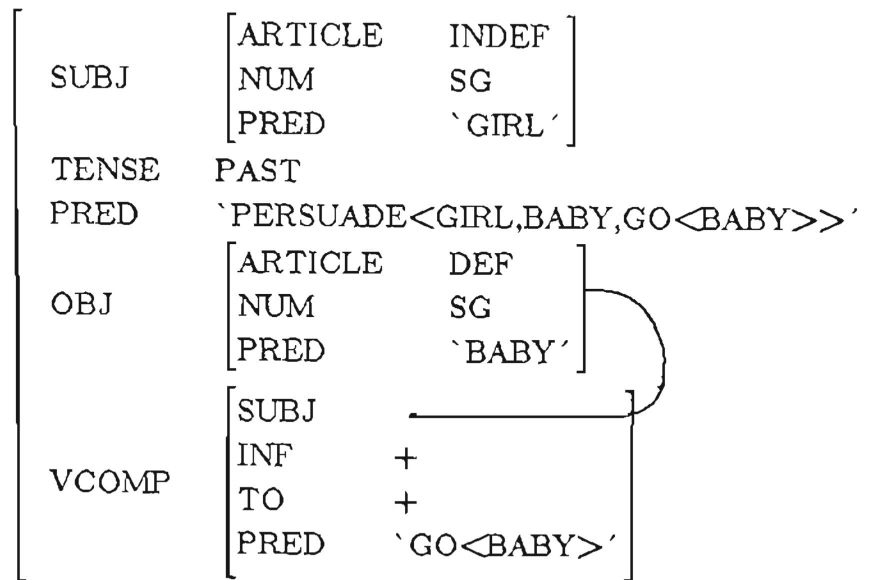

Figure 2.2: An F-Structure Containing Shared Attributes

are shown directly beneath non-terminals on the right side of the context-free rules.

The analysis of an input proceeds by first finding a context-free analysis of the sentence. In the second step, an single f-structure will be associated with each node in the parse tree. Initially, these f-structures are unknown and are represented by variables. Next, the f-description is constructed by instantiating the equation schemata to yield a set of equations over these unknown f-structures. Finally, the equations are solved by performing a unification for each equation in the f-description.

S	→	NP (↑ SUBJ) = ↓	VP ↑ = ↓	
NP	→	DET N		
VP	→	V	NP (↑ OBJ) = ↓	NP (↑ OBJ2) = ↓

a:	DET	(↑ DEF-INDEF) = INDEF (↑ NUM) = SG
girl:	N	(↑ NUM) = SG (↑ PRED) = 'GIRL'
handed:	V	(↑ TENSE) = PAST (↑ PRED) = 'HAND<(↑ SUBJ) (↑ OBJ) (↑ OBJ2)>'
the:	DET	(↑ DEF-INDEF) = DEF
baby:	N	(↑ NUM) = SG (↑ PRED) = 'BABY'
toy:	N	(↑ NUM) = SG (↑ PRED) = 'TOY'

Figure 2.3: A Simple Lexical-Functional Grammar

An up-arrow (↑) in an equation schemata refers to the structure for the non-terminal on the left side of the rule. A down-arrow (↓) refers to a structure associated with a non-terminal of the right side of the grammar rule. These arrows indicate *immediate domination*. For example, the equation

$$(\uparrow \text{SUBJ}) = \downarrow$$

in the first rule says that the value of the *SUBJ* feature of the f-structure associated with the S constituent is equivalent to the f-structure associated with the NP constituent. The terminals in the lexicon are handled similarly, with equation schemata containing only up-arrows.

Each side of an equation can either name a part of an f-structure or provide a constant value. Structure locations are given using feature names or sequences of feature names. These sequences are called ***path names*** and are given relative to the functional structure associated with the non-terminals of the grammar rule. In several of the schemata associated with lexical entries, quoted constant values appear. The quotes are used to indicate semantic translation forms. This grammar translates the verb ***handed*** using a three-placed predicate *HAND*. For instance, this grammar translates the sentence ***The girl handed the baby a toy.*** into

HAND<GIRL, TOY, BABY>

Note that the semantic forms appearing in the equation schemata are not atomic but can contain path names. The second schema associated with ***handed*** is an example.

The equations are processed sequentially to incrementally build up the functional structures. The processing of a rule can affect the (incomplete) functional structure in several ways. It can equate different feature values (i.e., bind them as with variable binding in Prolog), add in new values where there were previously none, create new

functional structures (by providing functional structures as literals in the equation) and, finally, disallow particular grammar interpretations by trying to equate two conflicting feature values. All this processing is just specializations of unification. In a valid solution for an f-description, the f-structure associated with the root non-terminal is considered the interpretation of the sentence.

To illustrate the use of the LFG shown in Figure 2.3, look at Figure 2.4, which shows a context-free parse of *The girl handed the baby a toy*. Functional structures labeled *x1*, *x2*, *x3*, *x4*, and *x5* are associated with the interior nodes in the parse tree and Figure 2.5 shows the equations with these names substituted in place of the arrows. (These instantiated equations constitute the functional description.) Finally, Figure 2.6 shows a (minimal) solution for these equations. The minimal solution is unique, if it exists. The functional structure *x1* (associated with the root non-terminal S) is the result of the LFG parse.

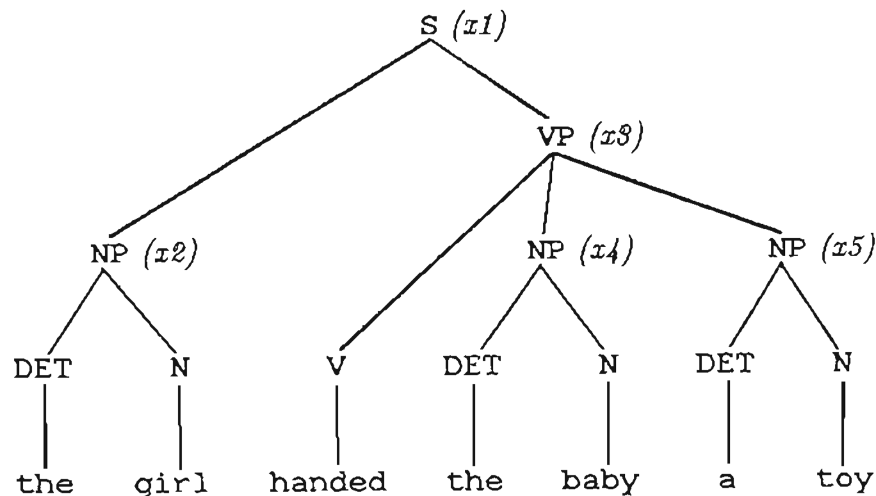

Figure 2.4: A Context-Free Parse

Rule or Word	Instantiated Equations
S → NP VP	(x1 SUBJ) = x2
	x1 = x3
VP → V NP NP	(x3 OBJ) = x4
	(x3 OBJ2) = x5
the	(x2 DEF-INDEF) = DEF
girl	(x2 NUM) = SG
	(x2 PRED) = 'GIRL'
handed	(x3 TENSE) = PAST
	(x3 PRED) =
	'HAND<(x3 SUBJ) (x3 OBJ) (x3 OBJ2)>'
the	(x4 DEF-INDEF) = DEF
baby	(x4 NUM) = SG
	(x4 PRED) = 'BABY'
a	(x5 DEF-INDEF) = INDEF
	(x5 NUM) = SG
toy	(x5 NUM) = SG
	(x5 PRED) = 'TOY'

Figure 2.5: The Instantiated Equations

$$x1{=}x3{=}\begin{bmatrix} \text{SUBJ} & \text{x2} \\ \text{OBJ} & \text{x4} \\ \text{OBJ2} & \text{x5} \\ \text{TENSE} & \text{PAST} \\ \text{PRED} & \text{`HAND<GIRL,BABY,TOY>'} \end{bmatrix}$$

$$x2{=}\begin{bmatrix} \text{DEF-INDEF} & \text{DEF} \\ \text{NUM} & \text{SG} \\ \text{PRED} & \text{`GIRL'} \end{bmatrix}$$

$$x4{=}\begin{bmatrix} \text{DEF-INDEF} & \text{DEF} \\ \text{NUM} & \text{SG} \\ \text{PRED} & \text{`BABY'} \end{bmatrix}$$

$$x5{=}\begin{bmatrix} \text{DEF-INDEF} & \text{INDEF} \\ \text{NUM} & \text{SG} \\ \text{PRED} & \text{`TOY'} \end{bmatrix}$$

Figure 2.6: The Solutions

In addition to providing context-sensitive constraints to weed out the ungrammatical interpretations, the equations are also used to add semantic structures to represent the meaning of the sentence. The equations of Figure 2.5 are designed to build the predication

$$\text{Hand} < \text{Girl, Baby, Toy} >$$

to represent the sentence's meaning. However, the resulting functional structure will be highly annotated with extra-syntactic information from the parse and is thus in a form convenient for further domain-specific processing when such a simple translation scheme is inadequate.

The paragraphs above describe what we will call the LFG *core*. Next, we describe additional aspects and features of LFG.

So far, the path names in the equations have been linear sequences of feature names. To handle the long distance dependency associated with left extraposition, we often need to do some (simple) searching in the parse tree to locate the appropriate constituent to use in an equation.

An example of a left extraposition occurs in the sentence *Which man did you want Sarah to give the book to?* We want to use the same grammar rules in parsing this sentence as in parsing *Sarah gives the book to the man*. Since this transformation (picking a noun phrase out of a declarative clause, changing its form slightly, and moving it to the beginning of the clause) can happen with many different forms of clauses, we need a way to parse long distance dependencies without increasing the complexity of the grammar. An typical LFG handles left extraposition by including a grammar rule to parse a null (zero length) noun phrase called the *trace*. In the example, *Which man did you want Sarah to give the book to?*, the trace occurs at the end of the sentence. The equations

associated with the null noun-phrase rule must link (unify) the extraposed noun phrase (*which man*) into the appropriate slot of the structure for the declarative clause containing the hole[2]. To perform this linking, we need a rule that says something like "Look at the structure beneath the verb phrase for a noun phrase structure that has a feature value of *Null* (i.e., the trace position). Fill in information from the questioning noun phrase found at the beginning of the sentence." The null noun phrase can be just about anywhere. (Consider *Which book did you want Sarah to give [trace] to the man?*) To perform this type of searching, a special variety of up- and down-arrows, called *bounded-domination*, is introduced. The bounded-domination up-arrow searches up the parse tree looking for a matching bounded-domination down-arrow. This long-distance linking performs the identification of the trace and the marker, completing the handling of extraposed constituents in LFGs.

In the LFG core, we saw only one kind of equality equation, which corresponds to the equality of unification. Other kinds of equations are allowed. In a *constraint equation*, the equality of attributes is tested. But unlike the equality equation, the attributes must already be bound: such an equation is not allowed to instantiate variables. A *negative equation* requires two attributes to be distinct without saying what they are. An *existential equation* requires that an attribute exist and be instantiated without providing its actual value. In the presence of constraint, negative, and existential equations, the equality equations are all processed first to bind the variables through unification. Then, in a second pass, the remaining equations are checked.

Several "conditions" place restrictions on what is to be considered an acceptable f-structure solution for a set of equations and hence what constitutes a legal LFG

[2] This linking is exactly what Extraposition Grammars (see Chapter 1) do in the DCG framework.

interpretation. To understand these conditions, we need to define *governable grammatical functions.* The predicate forms appearing in the equation schemata in the lexical entries make reference to certain attributes in the f-structure to be built for an input. For example, the equation in the lexicon entry for *handed*

$$(\uparrow \text{Predicate}) = \text{'HAND} < (\uparrow \text{SUBJ})\ (\uparrow \text{OBJ})\ (\uparrow \text{OBJ2}) >\text{'}$$

references the attributes *SUBJ, OBJ,* and *OBJ2.* An attribute (such as *SUBJ*) is said to be a *governable grammatical function* if it appears in the predication of any lexical entry.

The *completeness condition* requires an f-structure solution to contain all attributes that are mentioned in any semantic predication forms. That is, the semantic predicate's slots have to be filled in. For example, if a sentence's f-description included the above equation for *handed,* then any f-structure solution had better contain the features *SUBJ, OBJ,* and *OBJ2.* The *coherence condition* stipulates that the f-structure contain no governable grammatical functions that are not used in the semantic predications i.e., no leftovers in the sentence that don't participate in the meaning. Finally, the *grammaticality condition* says that an input is grammatical only if it is assigned a unique f-structure that is both complete and coherent[3].

Two other enhancements to the LFG core can be illustrated by the handling of prepositional phrases. In the core, feature labels were presented as atomic symbols that were specified literally in the equation schemata and that were not the subject of computation. However, equations such as

$$(\uparrow (\downarrow \text{PREP-CASE})) = \downarrow$$

[3] Note that the uniqueness requirement means that every ambiguous sentence is ungrammatical.

are allowed. This equation would be used in a rule that allows the attachment of prepositional phrases to a clause. The down-arrow refers to the f-structure associated with the prepositional phrase, the up-arrow with the clause. This equation says: Make the prepositional phrase's f-structure the value of one of the clause f-structure's attributes. Which attribute? The attribute label that also occurs as the *value* of the *PREP-CASE* attribute of the prepositional phrase's f-structure. For example, suppose the prepositional phrase is attached with the preposition *to* and that its f-structure has a feature called *PREP-CASE* with a value of *to*. This equation then causes the value of the *to* feature of the clause's f-structure to be set to the prepositional phrase's f-structure. Thus, the set of feature labels and the set of atomic feature values are not really distinct and incomparable. This type of equation is analogous to indirect addressing, where numbers are treated as both data and addresses of data.

Next, consider how multiple prepositional adjuncts, as in *The girl handed the baby a toy on Tuesday in the morning*, can be handled. We mentioned that the value of a feature could be atomic, a nested f-structure, a predication, or a sets of f-structures. Sets would be used here where there is no way to predict how many adjuncts may be present. The grammatical rule in question[4]

$$\mathrm{VP} \rightarrow \mathrm{V} \qquad \underset{(\uparrow \mathrm{OBJ})=\downarrow}{\mathrm{NP}} \qquad \underset{(\uparrow \mathrm{OBJ2})=\downarrow}{\mathrm{NP}} \qquad \underset{\downarrow\in(\uparrow \mathrm{ADJUNCTS})}{\mathrm{PP}^*}$$

contains a set membership ($\in$) equation.

Lexical-Functional Grammars were so named because they tend to place a strong emphasis on the lexicon. Since each word entry contains semantic information as well as extensive syntactic usage information (all expressed as equations), the dictionary

[4] Note that the Kleene star is allowed in LFG rules.

tends to be large, containing many similar entries. Moving syntactic information down into each word entry that can use it makes it easier to handle exceptions and special cases, reducing the size of the grammar rules, but results in redundancy in the storage of grammatical knowledge. Since most words can be used many ways and each requires a separate entry, there is a profusion of dictionary entries. For example, it is difficult to handle the passive transformation with LFG rules. Instead, separate entries are included in the lexicon for the active and passive senses of each verb.

In order to capture such regularities with any elegance, *lexical redundancy rules* are used to generate additional word entries given base entries. The following rule, for example, would be used to automatically generate the entry for the passive sense of a verb given the active sense.

$$(\uparrow \text{OBJ2}) \Rightarrow (\uparrow \text{OBJ})$$
$$(\uparrow \text{OBJ}) \Rightarrow (\uparrow \text{TO-OBJ})$$

Consider the lexical entry for *handed* shown in Figure 2.3. This lexical transformation rule automatically adds a second entry to the lexicon:

handed: V $(\uparrow \text{TENSE}) = \text{PAST}$
$(\uparrow \text{PRED}) = \text{'HAND}<(\uparrow \text{SUBJ})\ (\uparrow \text{TO-OBJ})\ (\uparrow \text{OBJ})>\text{'}$

2.2.1. Functional Unification Grammars

In work closely related to the development of Lexical Functional Grammars, Martin Kay developed *Functional Unification Grammars* (FUG) based on earlier work on *Functional Grammars* and *Unification Grammars* [Kay 1979, 1984a, 1984b, 1985]. In FUG, the grammar is itself expressed with functional structures. Instead of processing individual equations, unification is done between the structures representing the grammar. Certain features containing information about the context-free component of the

grammar are recognized as predefined. Special treatment of these features embeds the context-free parse into this framework.

In addition to the definition of f-structure given above, *disjunctions* of f-structures are needed to express alternatives in the grammar (e.g., alternate rules for expanding a NP). Other aspects of FUG allow the grammar to specify negative information (e.g., feature *f* cannot have a value) and required information (e.g., feature *f* must have a value).

In most variations of the formalism, every functional structure contains a feature named *Category* whose value, which is always an atomic symbol, names the structure. The name given to a structure is the same as the grammar non-terminal with which it is associated. A second feature called *Patterns* (sometimes *Constituent-Structure* or *Constituent-Set*) is a list containing information about what syntactic features are present in the structure. The order of features is unimportant but these two generally appear first. The special handling of these two features within the unification algorithm allows context-free parsing to be incorporated into the unification of the f-structures representing the grammar rules.

2.3. PATR-II

PATR-II grew out of work with several other linguistic formalisms, including Lexical-Functional Grammar (LFG), Functional Unification Grammar (FUG), Generalized Phrase Structure Grammar (GSPS), Head-driven Phrase Structure Grammar (HPSG), and Definite Clause Grammar (DCG). PATR-II was developed as a linguistic tool rather than as a linguistic theory and, as such, the emphasis was on increased expressiveness instead of restrictive expressiveness [Shieber 1984, 1985a].

The data structure of PATR-II, the feature structure (or f-structure), is identical to the f-structure from LFG with only slight notational differences [Karttunen 1984]. PATR-II also allows shared (i.e., *identical*) values within a structure, which are called *reentrant* values and which are indicated using coindexing boxes as shown in Figure 2.7. Feature structures are most easily understood by viewing them as directed, acyclic[5] graphs (DAGs) with a single root. The arcs are annotated with feature labels and the leaves are annotated with atomic feature values. For example, the f-structure of Figure 2.7 is shown as a DAG in Figure 2.8.

A partial order, called *subsumption*, is defined among f-structures. When f-structure A subsumes f-structure B, we say that A is *more general* than B and that B is *more specific* than A. In summary, f-structure A subsumes f-structure B if, for every feature f occurring in A, f also occurs in B and the value of f in A subsumes the value of f in B. When coindexing occurs in B it must be at least as restrictive as the coindexing occurring in A. If A subsumes B we write $A \sqsubseteq B$[6].

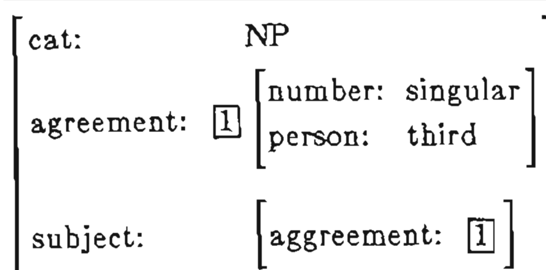

Figure 2.7: A Reentrant F-Structure in PATR-II Notation

[5] Some implementations also accommodate cyclic f-structures

[6] When Inheritance Grammar is presented in Chapters 3 and 4, we will provide a more rigorous definition of subsumption that generalizes the notion of subsumption used by PATR-II. There, we will use an order ($\sqsubseteq$) opposite to the order used in PATR-II. When X subsumes Y, we will write $Y \sqsubseteq X$ while the PATR-II notation writes $X \sqsubseteq Y$.

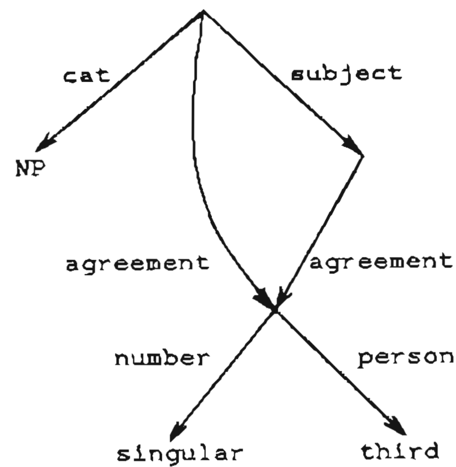

Figure 2.8: The DAG representation of an F-Structure.

The *unification* of two f-structures, A and B, is defined formally as the most general f-structure, C, such that $A \sqsubseteq C$ and $B \sqsubseteq C$. If such a f-structure C exists, it is unique. If no such f-structure exists, the unification is said to fail. *Generalization* is defined similarly, as a dual to unification.

In a PATR-II analysis, a substring of words in the sentence can be associated with an f-structure. The grammar rules describe how to combine substrings to produce larger strings and how to combine the associated f-structures to produce the more detailed f-structure associated with the larger strings. Concatenation is used as the string combining operation and unification is used as the f-structure combining operation.

As an example, consider the following PATR-II grammar rule.

X → Y Z

<X cat> = S
<Y cat> = NP
<Z cat> = VP
<X head> = <Z head>
<X head subject> = <Y head>

The first line of the rule is a context-free grammar rule. The remaining lines each specify a single unification. In these unification "equations," positions within particular f-structures are given using *paths*. This rule says that a string Y can be concatenated to a string Z to form a valid constituent X if the f-structures associated with X, Y, and Z can be unified according to the equations. For example, the last equation says that the *subject* attribute of the *head* attribute of the f-structure associated with X must unify with the *head* attribute of the Y structure.

Most unification-based linguistic theories use a *cat* feature in every f-structure associated with a syntactic constituent with the name of that constituent as its value, as is done by these equations. PATR-II includes a little notational sugar to make the expression of PATR-II rules more palatable. This same rule can be written as:

S → NP VP

<S head> = <VP head>
<S head subject> = <NP head>

The lexicon associates f-structures with words. In the textual specification of the lexicon, the grammar writer provides a set of equations for each word. It is the equations that specify the f-structure to be associated with the word. All paths in the equations for word W are given relative to the root of the f-structure to be associated with word W. Solving them builds the desired DAG. Consider the word entry for *John*:

```
Word john:
    <cat> = NP
    <head agreement gender> = masculine
    <head agreement person> = third
    <head agreement number> = singular
```

There is a great regularity among the words and their lexical entries. There are a lot of masculine, third-person, singular proper nouns. To build a dictionary without having to repeat each of these features for every word, there is a facility for grouping words and repeating common attributes. For example, suppose we wish to create a category for singular nouns called *SingNP*. This category is created with the following declaration:

```
Let SingNP be
    <cat> = NP
    <head agreement number> = singular
```

We can then instantiate a word using this category and possibly adding additional features as follows:

```
Word john: SingNP
    <head agreement person> = third
    <head agreement gender> = masculine
```

Hierarchies of word orders can also be accommodated. Another way to define the word *John* is to create a kind of a *SingNP* for third-person, singular noun phrases as follows:

```
Let SingNP be
    <cat> = NP
    <head agreement number> = singular

Let ThirdSingNP be SingNP
    <head agreement person> = third

Word john: ThirdSingNP
    <head agreement gender> = masculine
```

Multiple Inheritance is also accommodated. Supposed we also had a category for masculine words:

Let Masc be
<head agreement gender> = masculine

Then we could define *John* in yet a third way:

Word john: Masc ThirdSingNP

Classes, as well as instances of classes, are allowed to have multiple superclasses. There is also a notation to allow the overwriting of inherited features within subclasses.

Another technique to aid in constructing the dictionary is the use of Lexical Transformation Rules. These rules transform an existing f-structure (called the *in* structure) into a new f-structure (the *out* structure.) For example, a simple PATR-II transformation rule to generate the agentless passive form of a verb, given the active form of the verb, is

<out subj> = <in obj>
<out obj> = nil

We next make several comments about how the PATR-II notation is often used. First, lists can be represented in a way analogous to how they are represented in first-order logic. A list is represented with an f-structure (like a *cons*) whose *first* value is an element of the list and whose *rest* value is a list of the remaining elements. In PATR-II, an f-structure X representing a list of three elements (A, B, C) can be built with these equations:

<X first> = A
<X rest first> = B
<X rest rest first> = C
<X rest rest rest> = end

Second, lists are used extensively for verb subcategorization. Consider the following definition for the verb *likes* as in *Mary likes John*.

```
Word likes:
    <cat> = V
    <head form> = finite
    <syncat first cat> = NP
    <syncat rest first cat> = NP
    <syncat rest first head agreement person> = third
    <syncat rest first head agreement number> = singular
    <syncat rest rest> = end
```

This entry contains a feature called *syncat* whose value is a list. The elements of the list describe the verbal constraints on the noun phrase complements appearing in the sentence. The first element describes the object and the second element describes the constraints on the subject.

The relationship between the order of the list elements and the verb's complements is determined by the grammar rules, not shown here. Appendix 3 includes a complete PATR-II grammar in which this order is post-verbal complements in left to right order followed by the pre-verbal subject.

Third, f-structures can be used to represented the translation of an input sentence. When translating to first-order logic, we need to be able to represent first-order terms using f-structures. Typically, an f-structure with a *pred* feature is used to store the predicate (or functor) symbol and features *arg1*, *arg2*, etc. are used to store the arguments. For example, to build an f-structure X representing the predication p(a, b, c), the following equations could be used:

```
<X pred> = p
<X arg1> = a
<X arg2> = b
<X arg3> = c
```

This idea is easily extended for the representation of formulas containing logical

connectives (and, or, etc.) and quantifiers.

Lexical-Functional Grammar and PATR-II are obviously quite similar, at least when comparing the LFG *core* to PATR-II. Computationally, PATR-II is strictly more powerful. Parsing an input is semidecidable; any recursively enumerable language can be described. The offline parsability constraint[7] limits LFG to being decidable, although recognizing an input is NP-complete.

PATR-II is a declarative formalism while LFG is described in much more procedural terms, particularly the offline parsability constraint, the completeness and coherence conditions, bounded-domination, and negative, constraint and existential equations. Both top-down and bottom-up implementations exist for PATR-II including a PATR-II development environment called D-PATR, utilizing a chart-parsing approach [Karttunen 1986]. D-PATR runs on XEROX 1100 computers and provides a window/menu-based interface to perform and monitor incremental parsing. Hirsh has constructed a system to compile a PATR-II grammar into a Prolog program which uses a left-corner parsing strategy [Hirsh 1986]. A formal semantics of PATR-II, based on Dana Scott's domain theory has been developed [Pereira and Shieber 1984]. Additional work has addressed PATR-II implementation details such as structure-sharing and evaluation strategy [Karttunen and Kay 1985, Pereira 1986].

---

[7] This is the constraint, mentioned earlier, that says to parse an input using a LFG, we first find a context-free parse and then try to find a solution for the f-description

Part 2
Inheritance Grammar

Chapter 3: Introduction to Inheritance Grammar[1]

3.1. Introduction

Part One discussed the logic-based and unification-based approaches to NL processing. In Part Two, we introduce a new grammar formalism called *Inheritance Grammar* (IG). We have three goals for this new formalism. First, we want to incorporate feature-structures into a DCG-like framework. Second, we want to make semantic type checking more flexible than in the DCG approaches. Finally, we want the new formalism to be a proper superset of the DCG formalism. In this chapter, we informally introduce Inheritance Grammar, providing short examples to communicate its main characteristics and show how our objectives are met. A formal description of IG is given in the next chapter.

The IG formalism is an extension of Hassan Ait-Kaci's work on ψ-terms to the domain of grammatical analysis [Ait-Kaci 1984, Ait-Kaci and Nasr 1985]. A ψ-term is an informational structure similar to both the feature structure of LFG/PATR-II and the first-order term of logic. The set of ψ-terms is ordered by subsumption and forms a lattice in which unification of ψ-terms amounts to *greatest lower bounds* (GLB, $\sqcap$)[2]. In Inheritance Grammar, ψ-terms are incorporated into a computational paradigm similar

[1] Portions of this chapter appeared in [Porter 1987].

[2] Note that we identify unification with GLB rather than with LUB. In first-order logic and in PATR-II, unification is identified with LUB. This decision is fairly arbitrary since the development done here could be replaced by a dual development with unification as LUB. As we will soon see, unification of ψ-terms is based on an underlying IS-A taxonomy of symbols. Traditionally such IS-A taxonomies are oriented with superclasses *above* subclasses, motivating the sense we have chosen.

to the Definite Clause Grammar (DCG) formalism. Unlike feature structures and first-order terms, the atomic symbols of ψ-terms are ordered in an IS-A taxonomy, a distinction that is useful in performing semantic type-class reasoning during grammatical analysis. We begin by discussing this ordering.

3.2. The IS-A Relation Among Feature Values

Like other grammar formalisms using feature-based functional structures, we will assume a fixed set of atomic *symbols*. These symbols are the values used to represent lexical, syntactic and semantic categories and other feature values. This set is called the *signature*. In DCGs, it corresponds to the set of constants, functors, and predicate names. In many formalisms (e.g., DCG and PATR-II), equality is the only operation for symbols. In IG symbols are related in an IS-A hierarchy. These relationships are indicated in the grammar using statements such as:

```
boy < masculineObject.
girl < feminineObject.
man < masculineObject.
woman < feminineObject.
{boy, girl} < child.
{man, woman} < adult.
{child, adult} < human.
```

Symbols appearing in the grammar but not appearing in the IS-A statements are assumed to be unrelated. The symbol `<` can be read as "is a" and the notation $\{a_1, \ldots, a_n\} < b$ is an abbreviation for $a_1 < b, \ldots, a_n < b$. The grammar writer need not distinguish between instances and classes, or between syntactic and semantic categories when the hierarchy is specified. Such distinctions are only in the mind of the grammar writer and are reflected in the style with which the symbols are used in the grammar. Note that this example ordering exhibits multiple inheritance: `feminineObjects` are not necessarily `humans` and `humans` are not necessarily

`feminineObjects`, yet a `girl` is both a `human` and a `feminineObject`.

Computation of LUB ($\sqcup$) and GLB ($\sqcap$) in arbitrary partial orders is problematic. In IG, the grammar writer specifies an arbitrary partial ordering that the rule execution system automatically embeds in a lattice (an ordering in which LUB and GLB are always defined) by the addition of newly created symbols. The extension of the signature can be done without disturbing the order of the set.

What is the intuition behind the ordering statements and the lattice? Symbols may be thought of as standing for conceptual sets or semantic types and the IS-A relationship can be thought of as set inclusion. Finding the GLB – i.e., *unification* of symbols – then amounts to set intersection. For the partial order specified above, two new symbols are automatically added, representing semantic categories implied by the IS-A statements: i.e., human females and human males. The first new category (human females) can be thought of as the intersection of `human` and `feminineObject` or as the union of `girl` and `woman`[3], and similarly for human males. The ordering corresponding to the IS-A statements is shown in Figure 3.1 and the signature resulting from the embedding is shown in Figure 3.2. Our implementation automatically generates names – such as `{woman,girl}` – for the new symbols. If desired, the user can easily change these names to more informative strings.

3.3. Viewing ψ-terms as Enhanced Feature Structures

As we saw in the previous chapter, much work in computational linguistics is focussed around the application of unification to an informational structure that maps attribute names to values. A value is either atomic or (recursively) another such

[3] Or anything in between. One is the most liberal interpretation, the other the most conservative. The signature could be extended by adding both classes, and any number in between.

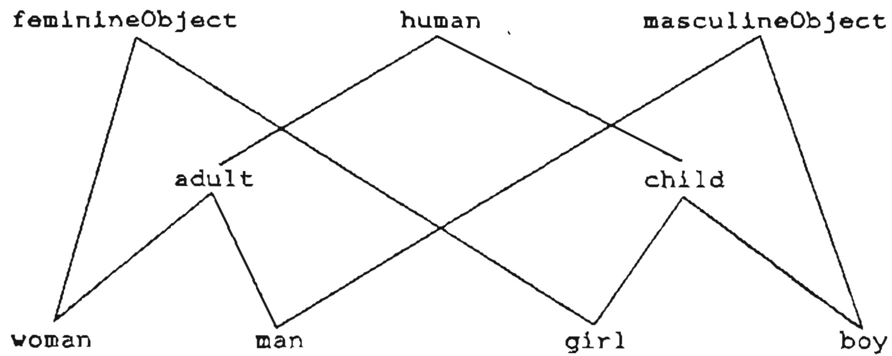

Figure 3.1: An IS-A Ordering with Multiple Inheritance

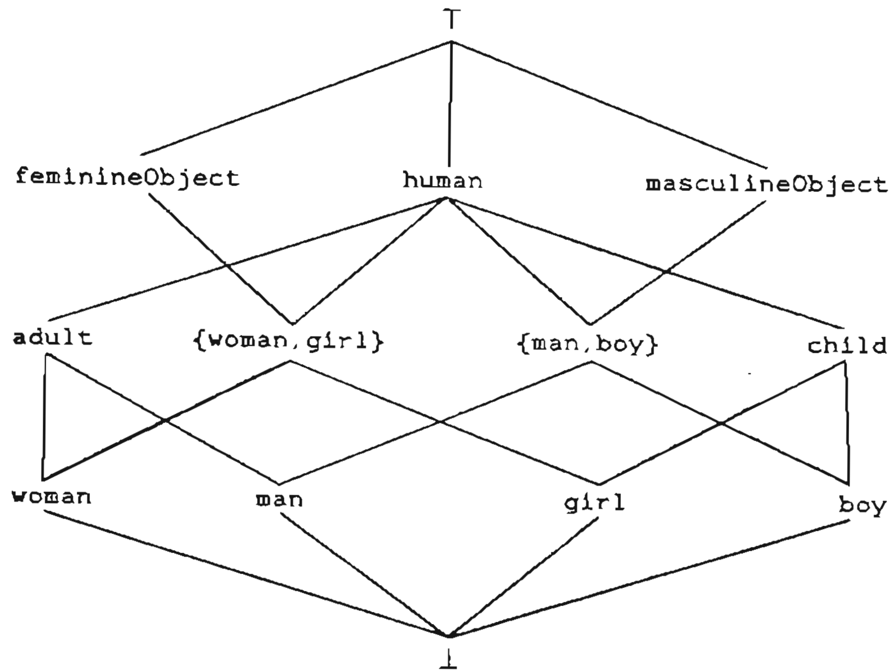

Figure 3.2: The Corresponding Signature Lattice

mapping. These mappings are called by various names: feature structures, functional structures, f-structures, and feature matrices. Recall the f-structures of PATR-II, which were described as rooted, directed, acyclic graphs (DAGs) whose arcs are annotated

with feature labels and whose leaves are annotated with atomic feature values.

IGs use ψ-terms, an informational structure that is best understood as a rooted, possibly cyclic, directed graph. Unlike in f-structures, every node (both leaf *and interior*) is annotated with a symbol from the signature. Each arc of the graph is labeled with a *feature label* (the analog of an *attribute*). The set of feature labels is unordered and is distinct from the signature. For the formal definition of ψ-terms, given in set theoretic terms, see the definition of the set Ψ in the next chapter. We will give several examples in this chapter to give the flavor of ψ-terms.

In addition to the DAG notation, feature structures are also represented using a bracketed matrix notation. We represent ψ-terms, on the other hand, using a textual notation similar to that of first-order terms. The syntax of the textual representation is given by the following extended BNF grammar. The characters ( ) $\Rightarrow$, and : are terminals. (Actually, this grammar is a slightly simplified version of the one presented in Chapter 4, which should be regarded as the complete description.)

TERM	::=	SYMBOL [ATTRIBUTELIST]
	|	ATTRIBUTELIST
ATTRIBUTELIST	::=	(ATTRIBUTE , ... , ATTRIBUTE)
ATTRIBUTE	::=	FEATURE $\Rightarrow$ TERM
	|	FEATURE $\Rightarrow$ VARIABLE [: TERM]

Our first example contains the symbols `np`, `singular`, and `third`. The label of the root node, `np`, is called the *head* symbol. This ψ-term contains two features, `number` and `person`.

```
np(number ⇒ singular,
   person ⇒ third)
```

The next example, illustrating nested structures, includes a subterm at `agreement⇒`

```
(cat        ⇒ np,
 agreement ⇒ (number ⇒ singular,
              person ⇒ third))
```

In this ψ-term the head symbol is missing, as is the head symbol of the subterm. When a symbol is missing, the most general symbol of the signature ($\top$) is assumed.

The location of a subterm within a ψ-term is given using a *path* (or *address*) relative to that ψ-term. For example, the subterm at address

```
agreement⇒number⇒
```

is the featureless ψ-term whose head is `singular`.

As Ait-Kaci has observed, a variable serves two purposes in traditional first-order terms [Ait-Kaci 1984]. First, as a wild card, it serves as a placeholder that will match any term. Second, as a constraint, one variable can constrain several positions in the term to be filled by the same structure. In ψ-terms, the wild card function is filled by the maximal symbol of the signature ($\top$), which will match any ψ-term during unification. Variables are used exclusively for the constraint function to indicate ψ-term *coreference*. By convention, variables always begin with an uppercase letter while symbols and labels begin with lowercase letters and digits.

In the following ψ-term, which can be thought of as a representation of the sentence *The man wants to dance with Mary,* `X` is a variable used to identify the subject of *wants* with the subject of *dance*.

```
sentence(subject   ⇒ X:man,
         predicate ⇒  wants,
         verbComp  ⇒  clause(subject    ⇒ X,
                             predicate  ⇒ dance,
                             prepObject ⇒ mary))
```

If a variable X appears in a term constraining a subterm t, then all subterms constrained by other occurrences of X must be consistent with (i.e., unify with) t. If a variable appears without a subterm following it, the term consisting of simply the *top* symbol ($\top$) is assumed. By convention, we prefer to write the subterm constrained by X following the first occurrence of X and not include subterms at all other occurrences of X. These secondary subterms then default to $\top$, which will trivially unify with the primary subterm at X. The constraint implied by variable coreference is not just equality of structure but equality of reference. Further unifications that add information to one sub-structure will necessarily add it to the other. Thus, in this example, `X` constrains the terms appearing at the paths `subject⇒` and `verbComp⇒subject⇒` to be the same term.

In the ψ-term representation of the sentence *The man with the toupee sneezed* shown below, the `np` filling the `subject` role, `X`, has two attributes. One is a `qualifier` filled by a `relativeClause` whose `subject` is `X` itself.

```
sentence(
  subject    ⇒ X: np(
        head       ⇒ man,
        qualifier ⇒ relativeClause(
              subject    ⇒ X,
              predicate ⇒ wear,
              object     ⇒ toupee)),
  predicate ⇒ sneezed)
```

As the graphical representation of this term in Figure 3.3 clearly shows, this ψ-term is cyclic.

3.4. Unification of ψ-terms

Before describing how ψ-terms are used in a grammar, we need to discuss *unification*, the operation with which we manipulate ψ-terms. The unification of two

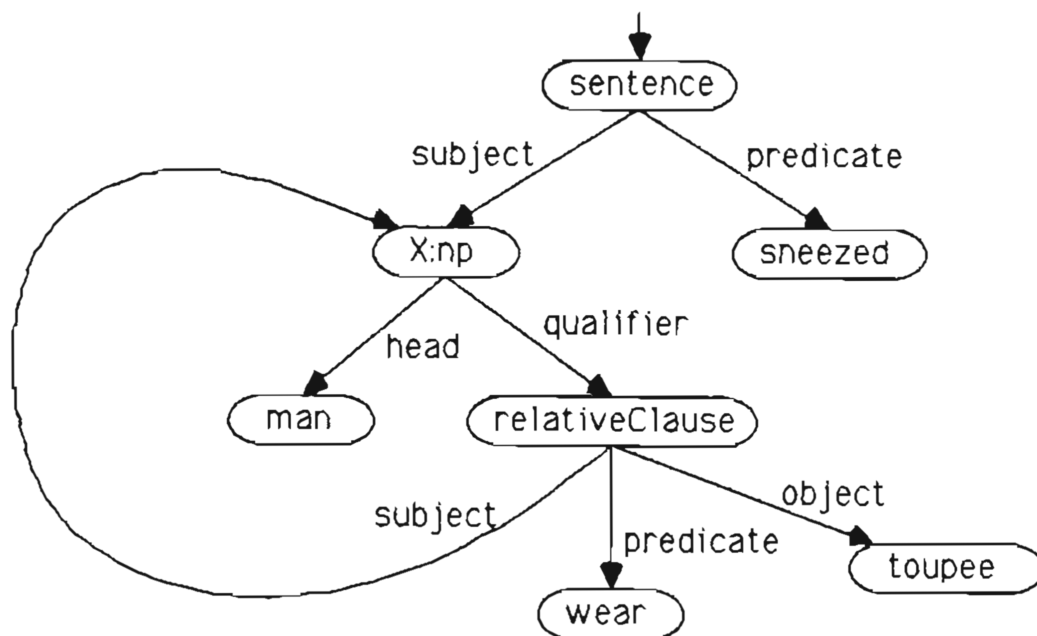

Figure 3.3: Graphical Representation of a Cyclic ψ-Term

ψ-terms is similar to the unification of two feature structures in PATR-II or two first-order terms in logic. Unification of two terms t_1 and t_2 proceeds as follows. First, the head symbols of t_1 and t_2 are unified. That is, the GLB of the two symbols in the signature lattice becomes the head symbol of the result. Second, the subterms of t_1 and t_2 are unified. When t_1 and t_2 both contain the feature f, the corresponding subterms are unified and added as feature f of the result. If one term, say t_1, contains feature f and the other term does not, then the result will contain feature f with the value from t_1. This result is the same that would obtain if t_2 contained feature f with value $\top$. Finally, the subterm coreference constraints implied by the variables in t_1 and t_2 are respected. That is, the result is the least constrained ψ-term such that if two paths in

t_1 (or t_2) are tagged by the same variable (i.e., they *corefer*) then they will corefer in the result.

As an example of unification (adapted from a similar PATR-II example [Shieber 1985a, page 19]), when the ψ-term

```
(agreement ⇒ X:(number ⇒ singular),
 subject   ⇒   (agreement ⇒ X))
```

is unified with

```
(subject ⇒ (agreement ⇒ (person ⇒ third)))
```

the result is

```
(agreement ⇒ X:(number ⇒ singular,
                person ⇒ third),
 subject   ⇒   (agreement ⇒ X))
```

As another example, assume we have an ordering including statements such as:

```
{smith, jones} < partTimeStudent.
{brown, anderson} < fullTimeStudent.
<partTimeStudent, fullTimeStudent} < student.
student < person.
{cse101, cse102, cse121} < course.
```

In the ψ-term

```
s(subject ⇒ X:student,
  verb    ⇒   takes,
  object  ⇒ Y:course)
```

X and Y function as *typed variables* when unified with another ψ-term, such as:

```
s(subject ⇒ smith,
  verb    ⇒ takes,
  object  ⇒ cse102)
```

The result of unifying these two ψ-terms is exactly the same as the second term:

```
s(subject ⇒ smith,
  verb    ⇒ takes,
  object  ⇒ cse102)
```

3.5. Relationship to F-structures and First-Order Terms

The f-structures of PATR-II and LFG can be viewed as a special case of ψ-terms, in which three conditions hold. First, the head symbol of all subterms (except those at the "lowest" level with no features) is $\top$. Second, the signature is a flat lattice, i.e., all symbols (besides $\top$ and $\perp$) are unordered. Third, in the case where cyclic f-structures are disallowed, the variable tagging of ψ-terms is constrained to be acyclic.

For example, the f-structure shown in Figure 3.4 is represented as the following ψ-term.

```
(subj ⇒ (det   ⇒ the,
         num   ⇒ plural,
         noun  ⇒ girls),
 vp   ⇒ (num   ⇒ plural)
         tense ⇒ present,
         verb  ⇒ run))
```

$$\begin{bmatrix} \text{SUBJ} & \begin{bmatrix} \text{DET} & \text{THE} \\ \text{NUM} & \text{PLURAL} \\ \text{NOUN} & \text{GIRLS} \end{bmatrix} \\ \text{VP} & \begin{bmatrix} \text{NUM} & \text{PLURAL} \\ \text{TENSE} & \text{PRESENT} \\ \text{VERB} & \text{RUN} \end{bmatrix} \end{bmatrix}$$

Figure 3.4: An F-Structure

The first-order term from logic can also be viewed as a special case of the ψ-term, again under three conditions. The symbols in the ψ-term correspond to the constants and functors of first-order terms. The first condition, that the signature again be a flat lattice, reflects the fact that the symbols in first-order terms are incomparable. The second condition reflects the fact that the arguments of a term are specified positionally and are not labeled. To represent ordered arguments, we restrict the feature labels of the ψ-term to be the features `1`, `2`, `3`, ... (In particular, the feature labels of the ψ-term must come from the set of the positive integers and if a feature n ($>$1) is present at an address, then feature n-1 must also be present.) Finally, we must constrain the ψ-term to reflect the fact that, in first-order terms, variables can only tag completely unbound subterms.

For example, the first-order term

```
f(a, X, g(b, X, c), Y)
```

would be represented as the ψ-term

```
f(1 ⇒ a,
  2 ⇒ X,
  3 ⇒ g(1 ⇒ b,
        2 ⇒ X,
        3 ⇒ c),
  4 ⇒ Y)
```

Unification of ψ-terms is very similar to unification of f-structures and of first-order terms. If A, B, and C are f-structures where C is the unification of A and B and where A′ and B′ are the corresponding representations of A and B as ψ-terms then C′ is the unification of A′ and B′, when C′ is the ψ-term representation of C. The same statement does not quite hold when A, B, and C are first-order terms. We will discuss the representation of first-order terms as ψ-terms further in the next chapter.

3.6. Inheritance Grammars

An IG consists of IS-A statements and *grammar rules.* A grammar rule is a definite clause that uses ψ-terms in place of the literals used in first-order logic programming. For example:

```
sent(mood ⇒ decl) :- np(number ⇒ X), vp(number ⇒ X).
```

Most of the notation of Prolog and DCGs is used. In particular, the `:-` symbol separates a rule head from the ψ-terms comprising the rule body. Analogously to Prolog, list notation (using `[`, `|`, and `]`) can be used as a shorthand for ψ-terms representing lists and containing `head` and `tail` features. When the `-->` symbol is used instead of `:-`, the rule is treated as a context-free grammar rule and the interpreter automatically appends two additional arguments (`start` and `end`) to facilitate parsing. The final syntactic sugar allows feature labels to be elided; numeric feature labels (i.e., `1`, `2`, `3`, ...) are automatically inserted before unlabeled attributes.

Our first simple Inheritance Grammar consists of the rules:

```
sent --> noun(number ⇒ Num),verb(number ⇒ Num).
noun(number ⇒ plural) --> [cats].
verb(number ⇒ plural) --> [meow].
```

The sentence to be parsed is supplied as a goal clause, as in:

```
:- sent([cats,meow],[]).
```

The parser first translates these clauses into the following equivalent IG clauses, expanding away the notational sugar, before execution begins.

```
sent(start ⇒ P1, end ⇒ P3) :-
        noun(number ⇒ Num, start ⇒ P1, end ⇒ P2),
        verb(number ⇒ Num, start ⇒ P2, end ⇒ P3).

noun(number ⇒ plural,
     start  ⇒ list(head ⇒ cats, tail ⇒ L),
     end    ⇒ L).

verb(number ⇒ plural,
     start  ⇒ list(head ⇒ meow, tail ⇒ L),
     end    ⇒ L).

:- sent(start ⇒ list(head ⇒ cats,
                     tail ⇒ list(head ⇒ meow,
                                 tail ⇒ nil)),
        end   ⇒ nil).
```

As this example indicates, every DCG is an Inheritance Grammar. However, since the arguments may be arbitrary ψ-terms, IG can also accommodate feature manipulation and taxonomic reasoning.

3.7. Search Strategy

The model-theoretic semantics of logic programming are given by defining logical implication in terms of validity relative to all possible interpretations. An actual implementation is complete if it is guaranteed to find a proof if the query is logically implied by the database clauses. The standard implementation, Prolog, is not complete. It uses a depth-first search strategy that can be implemented efficiently and gives the clauses a comforting procedural reading but that may fail to accurately capture logical implication by failing to find proofs. In this sense, Prolog is an inadequate implementation of logic programming.

Likewise, the formal semantics of Inheritance Grammar is specified in terms of models and logical implication. A particular implementation may or may not be complete with respect to this definition. As in logic, the depth-first strategy can be made

very efficient for IG but may fail to find solutions for some grammars.

The question of completeness is perhaps more important in the context of NL grammars than in other types of programs because left-recursion is more often a problem. As an example of left-recursive grammar rules, consider the following grammar fragment:

```
np --> modifiers, noun.
modifiers --> modifiers, adj.
modifiers --> det.
```

Obviously, this simple grammar can be rewritten to remove left-recursion and avoid infinite regress. However, an important goal in grammar formalisms is to make the formalism *completely* declarative, so that a grammar can be expressed in the clearest and "most declarative" way.

To illustrate, let us modify the rules so they construct a list of the modifiers:

```
np --> modifiers(L), noun.
modifiers([A|L]) --> modifiers(L), adj(A).
modifiers([D]) --> det(D).
```

These rules construct the list in a particular order. When the rules were rewritten in the obvious way to avoid left recursion, the list order is reversed:

```
np --> modifiers(L), noun.
modifiers([D|L]) --> det(D), adjlist(L).
adjlist(A|L]) --> adj(A), adjlist(L).
adjlist([]) --> [].
```

Again, we could obviously rewrite these rules to achieve the desired order, but only at the cost of increased grammar size. While this example is trivial, as the grammar grows to increase linguistic coverage, the amount of hacking that becomes necessary to accommodate the implementation also grows.

Complexity is a major problem in comprehensive grammars. Nontermination due to left-recursion is an artifact of a particular implementation, not a problem with the formalism itself. This observation holds for both DCGs and IGs equally.

3.8. Type-Class Reasoning in Parsing

Several logic-based grammars have used semantic categorization of verb arguments to disambiguate word senses and fill case slots (e.g., [McCord 1980, 1982]). One important motivation for using ψ-terms for grammatical analysis is to facilitate such semantic type-class reasoning during the parsing stage.

The approach taken in the McCord grammar is to use unification to do taxonomic reasoning. Two types unify if and only if one is a subtype of the other; the result is the most specific type. For example, if the term `smith:_`, representing an untyped individual, is unified with the type expression `X:(person:student)`, representing student (a subtype of person), the result is `smith:person:student`. Because of the way first-order unification is used to implement type unification, the type hierarchies must be tree-shaped.

We perceive two shortcomings to this approach. (1) The semantic hierarchy is somewhat inflexible because it is distributed throughout the lexicon, rather than being maintained separately. (2) Multiple Inheritance is not accommodated (although see McCord [1985]). In IG, the ψ-term `student` can act as a typed variable and unifies with the ψ-term `smith` (yielding `smith`) assuming the presence of IS-A statements such as:

```
student < person.
{smith, jones, brown} < student.
```

The taxonomy is specified separately — even with the potential of dynamic modification

— and multiple inheritance is accommodated naturally.

3.9. Other Grammatical Applications of Taxonomic Reasoning

The taxonomic reasoning mechanism of IG has applications in lexical and syntactic categorization as well as in semantic type-class reasoning. As an illustration, consider the problem of writing a grammar that accepts a prepositional phrase or a relative clause after a noun phrase but only accepts a prepositional phrase after the verb phrase. So *The flower under the tree wilted, The flower that was under the tree wilted,* and *John ate under the tree* should be accepted but not **John ate that was under the tree.*

A simple IG solution includes a taxonomy in which `prepositionalPhrase` and `relativeClause` are both `npModifiers` but only a `prepositionalPhrase` is a `vpModifier`. In the following highly abbreviated IG, when a `vpModifer` is called for after a `vp`, either the `prepositionalPhrase` rule or the `relativeClause` rule may be used, while only a `prepositionalPhrase` will do when an `npModifier` is called for.

```
{prepositionalPhrase, relativeClause} < npModifier.
prepositionalPhrase < vpModifier.

sent(...) --> np(...),vp(...),vpModifier(...).
np(...) --> np(...),npModifier(...).
np(...) --> ...
vp(...) --> ...
prepositionalPhrase(...) --> ...
relativeClause(...) --> ...
```

Since predicate names — e.g., `npModifier` — participate in the signature ordering, this example shows that ψ-terms are used at the predicate level, not just at the term level as was done in the LOGIN language [Ait-Kaci and Nasr 1986].

3.10. Implementation

We have implemented an IG development environment in Smalltalk on the Tektronix 4317 workstation to empirically investigate the usefulness of the formalism and have experimented with several grammars using this environment.

The IS-A statements are handled by an ordering package that performs the lattice completion dynamically by adding additional elements only when GLBs are requested. This package displays the signature graphically and allows interactive updating of the ordering.

Many of the techniques used in standard depth-first Prolog execution have been carried over to our IG execution environment, which we will describe in Chapter 5. To speed grammar execution, our system precompiles the grammar rules. To speed grammar development, incremental compilation allows individual rules to be compiled when modified.

As mentioned above, top-down, depth-first evaluation (which our implementation uses) is not *complete*. In Chapter 6, we will explore several complete evaluation strategies – including *Earley Deduction, Extension Tables,* and *Staged Depth-First Search Strategy* – that have been developed for first-order logic, showing how they can be adapted to Inheritance Grammars. Before discussing implementation, however, the next chapter presents a formal definition of IG, based on *ψ-logic.*

Chapter 4:
Definition of Inheritance Grammar

4.1. Introduction

In the previous chapter, we introduced Inheritance Grammar and provided some intuition for its formalization. In this chapter, we elaborate with several definitions and results. We begin with a description of ψ-terms, which are data structures similar to those in the work of Ait-Kaci [1984]. We provide a proof that the set of ψ-terms forms a lattice [Birkhoff 1979] and then we describe ψ-logic and ψ-resolution. Finally, we give a model-theoretic semantics for ψ-logic and Inheritance Grammars, defining *soundness* and *completeness* for proof procedures.

4.2. ψ-Terms

The atomic symbols that we will use as feature values comprise the *signature* of the grammar.

Definition. A *signature* (denoted Σ) is a finite lattice with $\top$ (top) and $\bot$ (bottom) elements. Symbols will be written as identifiers beginning with a lowercase letter or digit[1]. We will use the symbols $\leq$, $\bigwedge$, and $\bigvee$ for the lattice order, meet operation, and join operation, respectively. □

In Prolog, the head symbols of the terms `nounPhrase(X, Y)` and `properNounPhrase(X, Y)` are incomparable. Since it may be true that all proper

[1] Hereafter, the word "identifier" means a string of alphanumeric characters of arbitrary length ≥ 1.

noun phrases are noun phrases, we want these terms to be comparable. The grammar writer does so by specifying a signature in which `properNounPhrase < nounPhrase`. Since the signature is assumed to be predefined and fixed, we will take Σ as a given for the rest of this chapter. Later in this chapter we introduce ordering statements, which the grammar writer uses to specify a partial order. The implementation extends this order to a signature lattice.

The identifiers that will be used as feature labels comprise a set distinct (but not necessarily disjoint) from the signature. The next definition names this set.

Definition. The set of *features* (denoted **F**) is an unordered set of identifiers. □

In first-order terms, subterms are identified positionally. In ψ-terms, the sub-terms are identified using features. (We will use example ψ-terms to motivate our definitions although we will not formally define ψ-terms until later.) For example, the ψ-term

```
nounPhrase(determiner ⇒ the,
           number     ⇒ singular,
           head       ⇒ rock)
```

contains the features `determiner`, `number`, and `head`. In similar formalisms, features are called attributes, field names, labels, or slots.

Finally, we need some variables names. Variables will be used like Prolog variables to express equality constraints among sub-terms.

Definition. The set of *variables* (denoted **V**) is an unordered set of identifiers, each beginning with an uppercase letter. □

Example 1. The term

```
sentence(
     subject   ⇒ X: np(
               head      ⇒ man,
               qualifier ⇒ relativeClause(
                         subject   ⇒ X,
                         predicate ⇒ wear,
                         object    ⇒ toupee)),
     predicate ⇒ sneezed)
```

contains two occurrences of the variable X. □

The constraint that symbols begin with a lowercase letter and that variables begin with an uppercase letter will allow us to syntactically identify and distinguish between the symbols, features, and variables occurring in a given IG fragment.

Informally, a finite sequence of features in a ψ-term is called a *path* (or an *address*). Conventionally, the infix operator dot (.) is used to denote sequences. However, to make it clear that a sequence of identifiers is meant to be a path, we will also write a path as a sequence of features separated by the ⇒ symbol and add a final ⇒ after the last feature. Thus, the path `a.b.c` can be written more clearly as `a⇒b⇒c⇒`. We will overload the operator . to indicate concatenation of two paths, i.e., the appending of two feature lists.

Formally, we define an *address domain* **A** to be used to specify the addresses within a ψ-term. We will make the connection between address domains and ψ-terms as we define ψ-terms.

Definition. An *address domain* is a (possibly infinite) set of paths that is prefix-closed and finitely branching. More specifically, an address domain **A** is a set of feature sequences ($\mathbf{A} \subseteq \mathbf{F}^*$) such that:

(i) If l, m $\in \mathbf{F}^*$ and if l.m $\in$ **A** then l $\in$ **A** (prefix-closed), and

(ii) If l $\in$ **A** then $\{ f \mid f \in \mathbf{F}$ and l.f $\in \mathbf{A}\}$ is finite (finitely branching). □

The empty sequence ϵ is called the root address and is a member of every address domain. Addresses in **A** that are not prefixes of other addresses in **A** are called leaves.

Example 2. The address domain for the ψ-term

```
sentence(subj⇒john,
         vp  ⇒predicate(verb⇒run))
```

is {ϵ, `subj`, `vp`, `vp.verb`} or, equivalently, {ϵ, `subj⇒`, `vp⇒`, `vp⇒verb⇒`}. The leaves are `subj` and `vp.verb` or, equivalently, `subj⇒` and `vp⇒verb⇒`. □

An address domain may be infinite.

Example 3. The address domain

$$\mathbf{A}_1 = \{ \text{ s} \mid \text{s is a feature sequence of the form } \texttt{a}{\Rightarrow}(\texttt{b}{\Rightarrow}\texttt{a}{\Rightarrow})^* + (\texttt{a}{\Rightarrow}\texttt{b}{\Rightarrow})^* \}$$

is infinite. □

Definition. Given an address domain **A** and an address a $\in$ **A**, the address *subdomain* of **A** at a, denoted $\mathbf{A}\backslash a$ is $\{ x \mid x \in \mathbf{F}^* \text{ and } a.x \in \mathbf{A} \}$ □

Example 4. If $\mathbf{A}_2$ is the address domain {ϵ, `l`, `m`, `m.n`, `m.o`, `m.o.p`, `m.n.q`} then $\mathbf{A}_2\backslash$`m.o` = $\mathbf{A}_2'$ = {ϵ, `p`, `q`} is the address subdomain of $\mathbf{A}_2$ at `m.o`. The address domain $\mathbf{A}_2$ can be visualized as the tree in Figure 4.1 and $\mathbf{A}_2'$ can be visualized as the subtree in Figure 4.2. □

We next define regular address domains. To do this, we need the following definition.

Definition. Let subaddresses(**A**) denote the set of all address subdomains in **A**, i.e., subaddresses(**A**) = $\{ \mathbf{A}\backslash a \mid a \in \mathbf{A} \}$ □

A can be thought of as describing a tree and subaddresses(**A**) can be thought of as the set of all subtrees occurring in **A**.

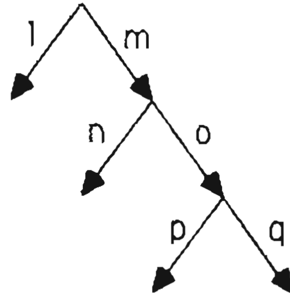

Figure 4.1: The address domain $\mathbf{A}_2$

Figure 4.2: The address domain $\mathbf{A}_2'$

Definition. An address domain $\mathbf{A}$ is *regular* (also called *rational*) if subaddresses(A) is finite[2]. □

Example 5. $\mathbf{A}_2$ (from Example 4) is regular since $\mathbf{A}_2$ is finite. $\mathbf{A}_1$ (also above) is an infinite address domain. Figure 4.3 shows an infinite tree representation of $\mathbf{A}_1$. $\mathbf{A}_1$ is regular since it has only a finite number of distinct subtrees:

$$\text{subaddresses}(\mathbf{A}) = \{ \mathbf{A}_1, \mathbf{A}_1' \}$$

[2] Colmerauer's concept of *rational trees* in the context of infinite Prolog terms is similar [Colmerauer 1986].

where

$$A_1' = \{ s \mid s \text{ is a feature sequence of the form } b{\Rightarrow}(a{\Rightarrow}b{\Rightarrow})^* + (b{\Rightarrow}a{\Rightarrow})^* \}$$

□

Example 6. Let $\mathbf{F} = \{0, 1, 2, ..., 9\}$ and $\mathbf{A}_3 = \{ n \mid n$ is a finite prefix of the digits in the decimal expansion of $\pi = 3.14159...\}$. $\mathbf{A}_3$ is not a regular address domain. □

Definition. A *directed graph*, or *digraph*, is a set of nodes and a set of edges connecting ordered pairs of nodes. In a *rooted* graph, one node is distinguished as the root node and all nodes are reachable from the root. The root is usually indicated with a

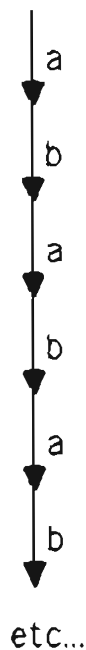

Figure 4.3: Infinite Representation of $\mathbf{A}_1$

small arrow pointing to it. A *finite* graph has a finite number of nodes and edges. □

Theorem 1. There is a one-to-one correspondence between regular address domains and finite, rooted digraphs with labeled edges. □

Proof. The proof is constructive: we provide a mapping from address domains to graphs and a mapping from graphs to address domains.

(i) For every address subdomain of the address domain **A** associate a node in a directed, rooted graph, G. The node associated with $\mathbf{A}\backslash\epsilon$ is the root. For every path l.m $\in$ **A**, where l $\in$ $\mathbf{F}^*$ and m $\in$ **F**, associate an edge directed from node $\mathbf{A}\backslash$l to node $\mathbf{A}\backslash$l.m. G is clearly finite. An inductive argument on path length shows that all nodes are reachable from from $\mathbf{A}\backslash\epsilon$.

(ii) Given a finite, rooted digraph G, let **F** be the set of features appearing in G. Associate a set of paths with each node in G as follows. Include ϵ in the set of path associated with the root. If path a $\in$ $\mathbf{F}^*$ is one of the paths associated with node n and there is an edge from n to node m labeled b, then include the path a.b in the set of paths associated with m. Let $\mathbf{A} = \{ a \in \mathbf{F}^* \mid a$ is associated with at least one node $\}$. Clearly **A** is prefix closed. **A** is finitely branching since each node in G has a finite number of edges. The number of subdomains of **A** is finite since we can associate each subdomain with a different node in G and there are finitely many nodes.

Since both mappings are injective, the correspondence is one-to-one. □

Since we are only interested in regular address domains, we imply regularity whenever we say *address domain* in what follows.

We are now ready to give our first cut at a definition of a *ψ-term*. We will refine it later.

Definition. Given a signature Σ, and set of features **F**, and a set of variables **V**, a ψ-term is a triple $<\mathbf{A}, \psi, \tau>$ where

(i) **A** is a regular address domain over **F**

(ii) ψ is a function that maps feature sequences (paths) into symbols, $\mathbf{F}^* \rightarrow \Sigma$, such that $\psi(a) = \top$ for all $a \notin \mathbf{A}$ and such that $\{ a \mid \psi(a) \neq \top \}$ is finite. That is, the symbol function (ψ) maps addresses into symbols and non-addresses into $\top$.

(iii) τ is a partial function mapping addresses into variables, $\mathbf{A} \rightarrow \mathbf{V}$, defined on only finitely many addresses. □

Example 7. Let t_1 be the triple $<\mathbf{A}_1, \psi_1, \tau_1>$, where $\mathbf{A}_1$ is listed in the first column of the following table and where the functions ψ_1 and τ_1 are given in the second and third columns. (All unlisted values of ψ_1 are of course $\top$.) Then t_1 is a ψ-term.

$\mathbf{A}_1$	ψ_1	τ_1
ϵ	sentence	X
subj	john	Y
verb	wanted	Z
verbCompl	predicate	V
verbCompl.subj	person	Y
verbCompl.verb	dance	W

□

We next describe two more-intuitive ways to represent a ψ-term. First, a ψ-term can be represented as a tree as follows. First construct the (possibly infinite) tree corresponding to the address domain **A**, labeling each arc with a feature from **F** such that each path from the root follows a sequence of features $a \in \mathbf{A}$. Then label each node with the variable given to that address by τ (if any) and with the symbol given by ψ. Throughout the rest of this chapter, we will abbreviate *ψ-term* to *term*. When we mean *first-order term*, we will say so explicitly.

Example 8. The term t_1 (from Example 7) is represented by the tree of Figure 4.4. Note that, at this point, we have no notion of coalescing of variables. □

Definition. A term is *finite* if its address domain is finite, and *infinite* otherwise. □

Finite terms are represented with finite trees and infinite terms with infinite trees.

Second, as demonstrated in preceding examples, regular ψ-terms (finite or infinite) can be represented textually using the syntax given by the following extended BNF grammar. The symbols ( ) $\Rightarrow$, and : are terminals.

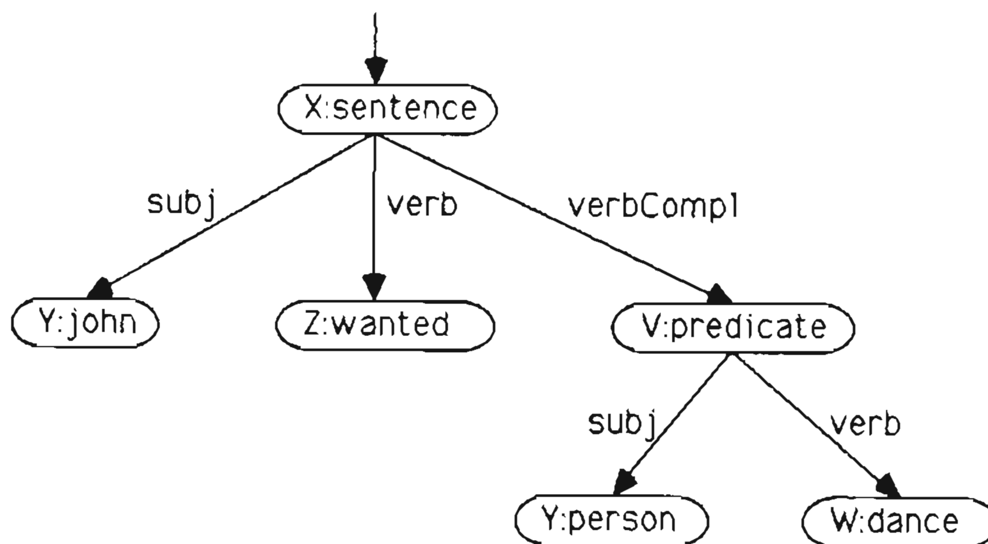

Figure 4.4: A Representation of t_1

```
TERM            ::=    [ VARIABLE : ] SYMBOL [ ATTRIBUTELIST ]
                    |  [ VARIABLE : ] ATTRIBUTELIST
                    |  VARIABLE
ATTRIBUTELIST   ::=    ( ATTRIBUTE , ... , ATTRIBUTE )
ATTRIBUTE       ::=    FEATURE ⇒ TERM
```

To summarize this grammar, a term is a symbol from the signature (called the *head* symbol), optionally followed by a parenthesized list of attributes. An attribute consists of a feature, followed by a term. A term may optionally be tagged with a variable as, for example, `X:nounPh`. Since – as we will see when alphabetic variants are discussed below – we are not too concerned with the actual variable names, we usually omit the variables that appear only once, regenerating them, when necessary, from previously unused variables such as X, Y, Z, ... When the head symbol is omitted, it is assumed to be $\top$. We will say more about variables when we discuss well-formedness below.

Example 9. The term t_1 (from Example 7) can be represented as:

```
X:sentence(
      subj      ⇒ Y:john,
      verb      ⇒ Z:wanted,
      verbCompl ⇒ V:predicate(
                          subj ⇒ Y:person,
                          verb ⇒ W:dance))
```

□

Example 10. By omitting variables that occur only once, we can write t_1 as:

```
sentence(
    subj      ⇒ Y:john,
    verb      ⇒   wanted,
    verbCompl ⇒   predicate(
                     subj ⇒ Y:person,
                     verb ⇒   dance))
```

Note that because redundant subterms can be elided and since $\top$ may be specified explicitly or implicitly, some terms have several textual representations. □

As in the previous example, a term often contains subterms. Each subterm is located at a specific address within the term. The next definition makes the notion of *subterm* more concrete.

Definition. Let $t = <\mathbf{A}, \psi, \tau>$. The *subterm of t at address a* (denoted $t\backslash a$), where $a \in \mathbf{A}$, is the term $<\mathbf{A}\backslash a, \psi\backslash a, \tau\backslash a>$ where:

$\mathbf{A}\backslash a$ is the subdomain of $\mathbf{A}$ at a,

$\psi\backslash a$ is a function $\mathbf{A}\backslash a \rightarrow \Sigma$ such that $\psi\backslash a(b) = \psi(a.b)$ for all addresses $b \in \mathbf{A}\backslash a$, and

$\tau\backslash a$ is the variable tagging function $\mathbf{A}\backslash a \rightarrow \mathbf{V}$ such that $\tau\backslash a(b) = \tau(a.b)$ for all addresses $b \in \mathbf{A}\backslash a$.

□

Example 11. The subterm of t_1 at `verbCompl⇒` is:

```
V:predicate(subj  ⇒ Y:person,
            verb  ⇒ W:dance)
```

□

Example 12. The subterm of t_1 at `verbCompl⇒verb⇒` is `dance`. □

Note that $t\backslash\epsilon = t$. That is, the subterm at the root of term t is term t itself.

Definition. Two addresses in a term bearing the same variable are said to *corefer*. □

Example 13. In t_1 above, the addresses `subj⇒` and `verbCompl⇒subj⇒` corefer. □

Definition. The *coreference relation* $\equiv$ on a term t is the equivalence relation relating exactly those addresses that corefer. □

In other words, the relation $a \equiv b$ is true for addresses a and b if and only if the variables at addresses a and b are the same. Any function, *f*, defined on set S induces an equivalence relation on S [Lipson 1981]; this is called the *kernel relation* induced by that function and is denoted *kernel(f)*. The coreference relation $\equiv$ is the kernel relation induced by τ and each coreference class contains coreferring addresses. That is, for a, b $\in \mathbf{A}$, we have $a \equiv b$ if and only if $\tau(a) = \tau(b)$. Clearly, given an equivalence relation $\equiv$, we can construct a function τ such that *kernel*(τ) is $\equiv$ since we have an infinite supply of variables.

The intent of variables is to force the subterms at different addresses to be identical. Since the definition of ψ-terms given above does not include this constraint, we next introduce the notion of well-formed terms.

Definition. A term $t = <\mathbf{A}, \psi, \tau>$ is *well-formed* if, for any two addresses a and b that corefer (i.e., $a \equiv b$), then, for all addresses $c \in \mathbf{F}^*$ such that $a.c \in \mathbf{A}$,

(i) $b.c \in \mathbf{A}$,

(ii) $\psi(b.c) = \psi(a.c)$, and

(iii) $\tau(b.c) = \tau(a.c)$ □

In other words, a term is *well-formed* if the same subterm occurs at all addresses that have the same variable. Since $a \equiv b$ implies $a.c \equiv b.c$ (for any addresses a, b, $c \in \mathbf{F}^*$ such that $a.c \in \mathbf{A}$ and $b.c \in \mathbf{A}$), $\equiv$ is a *right-invariant equivalence relation* (or *right-congruence*).

Example 14. Term t_1 is not a well-formed term since the addresses `subj⇒` and `verbCompl⇒subj⇒` corefer but different terms occur at `subj⇒` and at `verbCompl⇒subj⇒`. □

Example 15. The term t_2 (differing from t_1 only at `verbCompl⇛subj⇛`) shown below is well-formed:

A_1	ψ_1	τ_1
ϵ	`sentence`	X
`subj`	`john`	Y
`verb`	`wanted`	Z
`verbCompl`	`predicate`	V
`verbCompl.subj`	`john`	Y
`verbCompl.verb`	`dance`	W

This term can be represented textually as:

```
sentence(
     subj      ⇛ Y:john,
     verb      ⇛   wanted,
     verbCompl ⇛   predicate(
                       subj ⇛ Y:john,
                       verb ⇛    dance))
```

□

Since address domains for terms are regular, the following statements hold for every well-formed term, t.

(i) The number of subterms occurring in t is finite.

(ii) The number of symbols occurring in t (i.e., the number of addresses for which the value of ψ is non-$\top$) is finite.

(iii) The number of variables occurring in t is finite.

Henceforth, we shall only be concerned with well-formed terms. If a term t is well-formed, its textual representation is usually abbreviated by writing the subterm occurring at a set of coreferring addresses only once. Since the same term appears at each address in a set of coreferring addresses, the term only needs to appear after the first occurrence of the variable; all remaining addresses will just include the variable with the understanding that the shared term is meant. This abbreviation will allow

cyclic terms (defined below) to be represented textually.

For convenience, the implementation allows different but unifiable subterms to appear at corefering addresses. During parsing, these subterms are unified to yield the ultimate subterm used. (We will define ψ-term unification later.) If a variable X appears in one or more places in a textual representation, but no occurrences have an associated subterm, then the associated subterm is assumed to be $\top$, the most general term.

Example 16. Term t_2 from Example 15 is a well-formed term and hence could also be represented textually as:

```
sentence(
        subj       ⇒ Y:john,
        verb       ⇒   wanted,
        verbCompl  ⇒   predicate(
                              subj ⇒ Y,
                              verb ⇒ dance))
```

The subterm at address verbCompl⇒subj⇒ is john but it is left off since the variable Y appears elsewhere with the same subterm. □

Example 17. We can write the well-formed term

```
Z:f(k ⇒ X:g,
    l ⇒ Y:⊤,
    m ⇒ X:g,
    n ⇒ Y:⊤)
```

as

```
Z:f(k ⇒ X:g,
    l ⇒ Y,
    m ⇒ X,
    n ⇒ Y)
```

□

A well-formed term can be represented as a finite, rooted digraph. Just as in the tree representation, edges are annotated with features from **F**. Each node corresponds to an address and is annotated with the variable assigned to that address by τ, a colon, and the signature symbol assigned to the address by ψ. Furthermore, when two addresses corefer, the graph is drawn so that the shared subterm is represented only once. The root of the shared subterm is represented by a single node and edges corresponding to the coreferring addresses are all directed to that node. Since the connectivity of the graph captures effectively all the meaning the variables are meant to convey and since actual variable identifiers are often irrelevant identifiers such as X, Y, Z, ..., they are often left out of the graphical representation, especially for singleton address coreference classes.

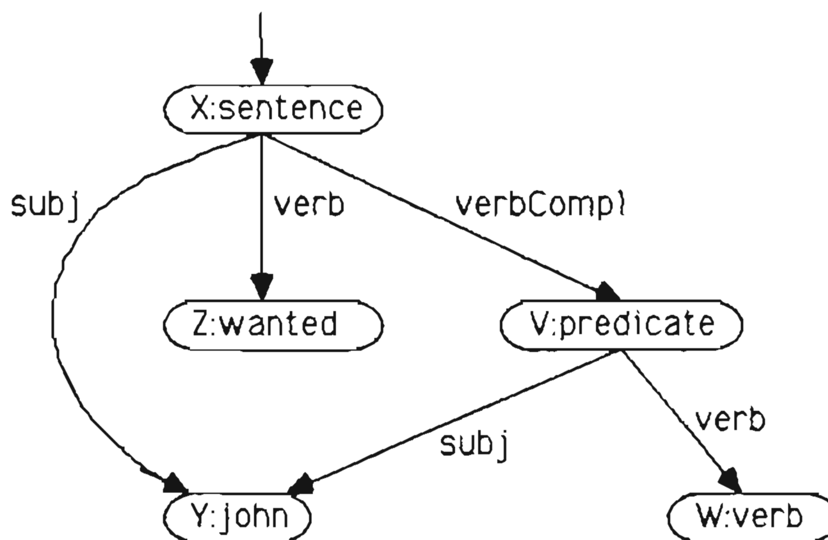

Figure 4.5: Graphical representation of t_2

Example 18. Figure 4.5 shows the graphical representation of term t_2 (from Example 15). Figure 4.6 shows t_2 with some variables omitted. □

Representing terms with graphs rather than trees makes it possible to represent infinite (but regular) terms by using cyclic graphs. We discuss the distinction between cyclic and acyclic terms next.

Definition. A variable X *occurs below* address a (written X **below** a) if X is in the codomain of $\tau\backslash a$. □

Example 19. In term t_2 (as given in Example 15 and Figure 4.5), we have W **below** ϵ, Y **below** verbCompl⇒, and, as a boundary case, Z **below** verb⇒. □

Definition. There is an *non-empty edge sequence* from address a to address b, written a **edge** b, if τ(b) **below** a.c for some $c \in \mathbf{F}$.

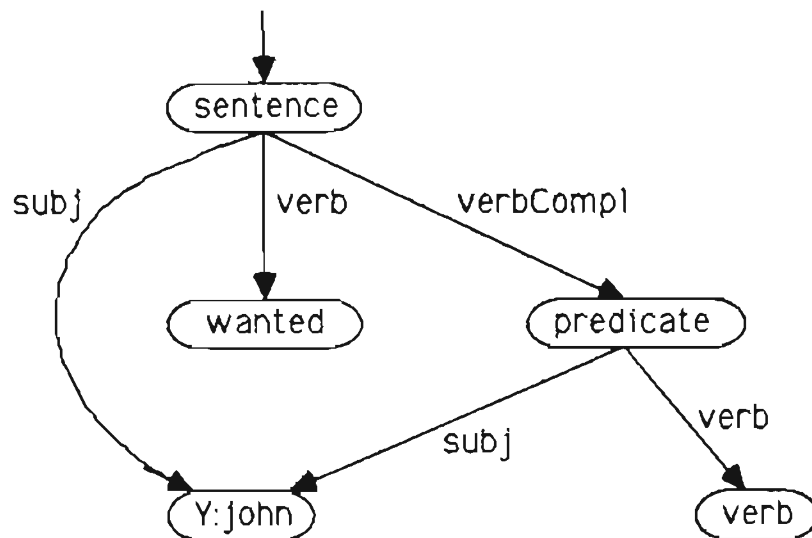

Figure 4.6: Term t_2 With Some Variables Elided

Definition. The **path** relation is defined as the transitive closure of **edge**. We use + to denote the transitive closure of a relation, so **path** = **edge+**. □

The relation **path** between addresses is true of two addresses, a **path** b, if there is a (non-empty) path in the graph representation of t from a to b.

Definition. A term is *acyclic* if **path** is irreflexive. All other terms are *cyclic*. □

Example 20. The term t_2 shown in Figures 4.5 and 4.6 is acyclic. □

Example 21. The following term, t_3, which might be a loose representation of *the man running the show*, is cyclic. It is shown in Figure 4.7.

```
X:nounPh(
        det      ⇒ the,
        head     ⇒ man,
        modifier ⇒ clause(
                        subj ⇒ X,
                        verb ⇒ run,
                        obj  ⇒ nounPh(
                                    det  ⇒ the,
                                    head ⇒ show)))
```

□

Since the purpose of variables is to show the connectivity structure of a term and we are not really concerned with the actual variable names used, we introduce the following definition.

Definition. Two terms are *alphabetic variants* if they are identical up to a one-to-one renaming of variables. More specifically, we write $t_1 \alpha t_2$ to mean $t_1 = \langle \mathbf{A}_1, \psi_1, \tau_1 \rangle$ is an alphabetic variant of $t_2 = \langle \mathbf{A}_2, \psi_2, \tau_2 \rangle$ if and only if

(i) $\mathbf{A}_1 = \mathbf{A}_2$,

(ii) $\psi_1 = \psi_2$, and

(iii) $\mathrm{kernel}(\tau_1) = \mathrm{kernel}(\tau_2)$. □

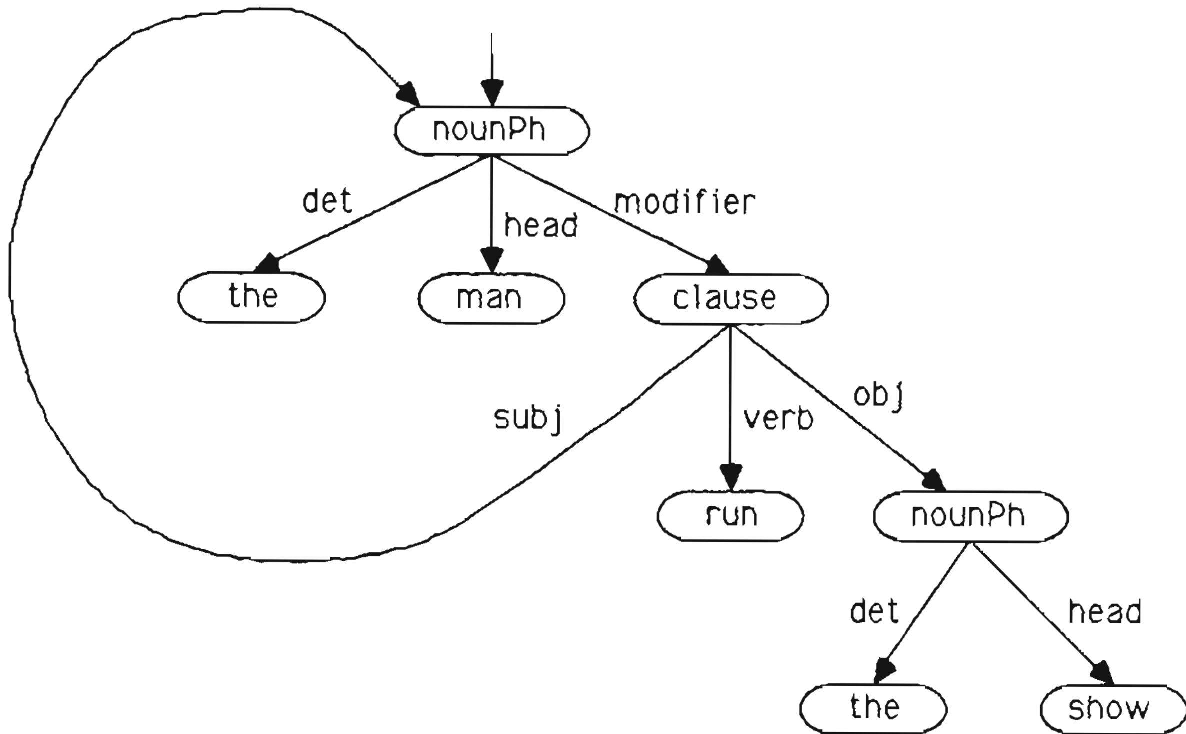

Figure 4.7: Graphical representation of t_3

Note that the graphical representation of a ψ-term in which the variables have been omitted is incapable of distinguishing alphabetic variants.

Definition. The term consisting of a single address, ϵ, and such that ψ maps that address to $\top$, the top element in the symbol signature, is called the *most general term* and is written textually as $\top$. □

Note that X, X: $\top$, and $\top$ are all perfectly legitimate representations of the most general term.

The $\perp$ symbol of the signature is to be considered as an inconsistent or error symbol. Since any term containing $\perp$ should also be treated as an error, i.e., as $\perp$ itself, we need the following definition.

Definition. The *error relation* equates two terms containing $\perp$. We write $t_1 \downarrow t_2$ if and only if $t_1=t_2$ or both t_1 and t_2 contain $\perp$. By "both contain $\perp$," we mean ψ_1 maps some address in A_1 to $\perp$ and ψ_2 maps some (possibly different) address in A_2 to $\perp$. □

The definition of ψ-terms used up to here (well-formed terms) is inadequate because it ignores variable renaming and error terms. However, it is close enough that it can easily be fixed by defining an equality among those terms we desire to be one-and-the-same. In the following definition of equality among ψ-terms, we ignore systematic renaming of variables (i.e., treat alphabetic variants as equal) and treat all error terms as equal.

Definition. Let $\approx$ be the relation relating alphabetic variants to each other and relating all error terms to $\perp$, i.e., let $\approx = (\alpha \cup \downarrow)$. □

Theorem 2. The relation $\approx$ is an equivalence relation. □

Proof. (original) We need to show that $\approx$ is reflexive, symmetric, and transitive. Reflexivity and symmetry follow from the reflexivity and symmetry of α and of $\downarrow$. To show transitivity, we need to show that if a $\approx$ b and b $\approx$ c then a $\approx$ c. The only non-trivial case is to show that a is related c when a is an alphabetic variant of b and when b and c both contain $\perp$. Since a is an alphabetic variant of b, it too contains $\perp$ and is thus related to c by $\downarrow$. □

Finally we are ready to define the set of all ψ-terms. We call this set Ψ.

Definition. Let $\mathbf{W}$ be the set of well-formed terms. Let Ψ be the quotient set of $\mathbf{W}$ modulo the equivalence relation $\approx$, i.e., $\Psi = \mathbf{W}/\approx$. □

Technically, the elements of Ψ are not ψ-terms, but equivalence classes of well-formed terms. However, we will represent each class by one of its members, t, rather than using the more formal [t]. Henceforth, by ψ-term we will mean elements of Ψ. The symbol $\perp$ will represent any and all terms containing $\perp$ anywhere, and it will be understood that the variables in a ψ-term may be (consistently) renamed without altering the identity of that ψ-term. In other words, two ψ-terms t_1 and t_2 are equal if and only if $t_1 \approx t_2$.

4.3. A Lattice of ψ-Terms

Using subsumption as an ordering, the set of first-order terms of logic forms a lattice. Is this property also true of Ψ, the set of ψ-terms? The next line of development shows that it is. We begin by defining three operators: subsumption ($\sqsubseteq$), generalization ($\sqcup$), and unification ($\sqcap$). Then we show that Ψ is a lattice with respect to subsumption ($\sqsubseteq$), that $\sqcup$ is the least upper bound (L.U.B.) operator, and that $\sqcap$ is the greatest lower bound (G.L.B.) operator.

Definition. A term $t_1 = <\mathbf{A}_1, \psi_1, \tau_1>$ is *subsumed* by another term $t_2 = <\mathbf{A}_2, \psi_2, \tau_2>$, written $t_1 \sqsubseteq t_2$, if $t_1 = \perp$ or

(i) $\mathbf{A}_2 \subseteq \mathbf{A}_1$ (Every address in t_2 is in t_1),

(ii) $\equiv_2 \subseteq \equiv_1$ (All addresses that corefer in t_2 also corefer in t_1; i.e., t_1 is more constrained), and

(iii) $\psi_1(a) \leq \psi_2(a)$ for all addresses $a \in \mathbf{A}_2$, where $\leq$ is the partial order on symbols (the symbol at any address in t_1 is less than the symbol at the corresponding address in t_2).

We also say that t_2 is *more general* than t_1. □

Technically, we have just defined $\sqsubseteq$ on well-formed terms but this definition extends naturally to ψ-terms, using $[t_1] \sqsubseteq [t_2]$ if and only if $t_1 \sqsubseteq t_2$. The choice of equivalence class representative is immaterial since subsumption is defined in such a way that variable renaming has no effect and since $\bot$ is subsumed by all terms.

Example 22. The ψ-term

```
nounPh(number⇒singular, head⇒fish)
```

is subsumed by

```
nounPh(head⇒fish)
```

□

Example 23. Assume that `singular < countable` in the signature Σ. Then

```
nounPh(number⇒countable, head⇒person)
```

is more general than

```
nounPh(number⇒singular, head⇒person)
```

□

Example 24. The term

```
f(l⇒X, m⇒Y)
```

subsumes

```
f(l⇒Z, m⇒Z)
```

since the second term has a more constrained variable tagging. □

Theorem 3. Subsumption of ψ-terms is a partial order. □

Proof. (original) We need to show reflexivity, antisymmetry, and transitivity.

First, reflexivity, $t \sqsubseteq t$, clearly holds.

Second, antisymmetry (if $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_1$ then $t_1 = t_2$) holds because the operators used in the definition of subsumption ($\leq$ and $\subseteq$) are antisymmetric.

Third, we need to show transitivity. Assume that $t_1 \sqsubseteq t_2$ and $t_2 \sqsubseteq t_3$. Obviously, $\mathbf{A}_2 \subseteq \mathbf{A}_1$ and $\mathbf{A}_3 \subseteq \mathbf{A}_2$ implies $\mathbf{A}_3 \subseteq \mathbf{A}_1$. Likewise, $\equiv_2 \subseteq \equiv_1$ and $\equiv_3 \subseteq \equiv_2$ implies $\equiv_3 \subseteq \equiv_1$. Finally, $\psi_1(a) \leq \psi_3(a)$ holds for all $a \in \mathbf{A}_3$ since $a \in \mathbf{A}_1$, $a \in \mathbf{A}_2$ and $\psi_1(a) \leq \psi_2(a)$ and $\psi_2(a) \leq \psi_3(a)$. □

Next, we define the generalization ($\sqcup$) and unification ($\sqcap$) operators. Viewing ψ-terms as representations or descriptions of sets of objects (from some domain – more on domains later), generalization corresponds to set union. Dually, unification can be understood as an intersection operator for set descriptions. Since the generalization operation has the simpler definition, we begin with it.

Definition. The *generalization* of two terms $t_1 = \langle \mathbf{A}_1, \psi_1, \tau_1 \rangle$ and $t_2 = \langle \mathbf{A}_2, \psi_2, \tau_2 \rangle$ (denoted $t_1 \sqcup t_2$) is the term $t = \langle \mathbf{A}, \psi, \tau \rangle$ where

(i) $\mathbf{A} = \mathbf{A}_1 \cap \mathbf{A}_2$

(ii) τ is a function in $\mathbf{A} \rightarrow \mathbf{V}$ such that kernel(τ) is $\equiv_1 \cap \equiv_2$, and

(iii) $\psi(a) = \psi_1(a) \vee \psi_2(a)$ for every address a in $\mathbf{A}$ and where $\vee$ is the L.U.B. operator for the signature. □

This definition says that (i) the only addresses in the generalization are those addresses that are in both t_1 and t_2, (ii) addresses in the generalization will corefer if and only if they coreferred in both t_1 and t_2, and (iii) symbols that occur at the same address in t_1 and t_2 are generalized.

This definition of generalization is based on class representatives, not equivalence classes. To show this definition is well-formed, we need to show that it is independent of the choice of class representatives, i.e., $t_1 \approx t_1'$ and $t_2 \approx t_2'$ implies $(t_1 \sqcup t_2) \approx (t_1' \sqcup t_2')$.

Since the definition is given in terms of coreference equivalence classes $\equiv_1$ and $\equiv_2$ and not of specific variable mapping functions τ_1 and τ_2, the choice of alphabetic variants is irrelevant. However, the definition says:

```
f(l⇒a) ⊔ f(l⇒ ⊥) = f(l⇒a)
```

while

```
f(l⇒a) ⊔ ⊥ = f
```

Since `f(l⇒a)` and `f` are not the same term, we must ammend the definition of generalization to stipulate that

$$[t] \sqcup [\bot] = [\bot] \sqcup [t] = [t].$$

Example 25. Assume that `employee < person` in the signature and that

```
t4 =
  employee(id   ⇒ name(last⇒smith, first⇒joe),
           dept ⇒ sales)
t5 =
  person(id ⇒ name(last⇒smith, first⇒fred)).
```

Then we have

$$t_4 \sqcup t_5 = \texttt{person(id} \Rightarrow \texttt{name(last}\Rightarrow\texttt{smith, first}\Rightarrow\texttt{X))}.$$

Informally, this example shows that the class of employees with first name "joe", last name "smith" and department named "sales" and the class of persons with first name "fred" and last name "smith" are generalized to the class of persons with last name "smith". In first-order logic, terms with distinct functors (such as `employee` and `person`) can not be generalized since there is no relation amongst functors other than equality. □

Example 26. Figure 4.8 shows two terms and their generalization, all represented graphically. In examining how coreference is treated in the ⊔ operation, notice how

only the common parts of the two argument graphs are included in their generalization. □

While generalization of two terms results in a term that essentially contains only the information that was in *both* argument terms, unification will produce a term containing the information that was present in *either* argument term. Before defining unification, we need some preliminary definitions.

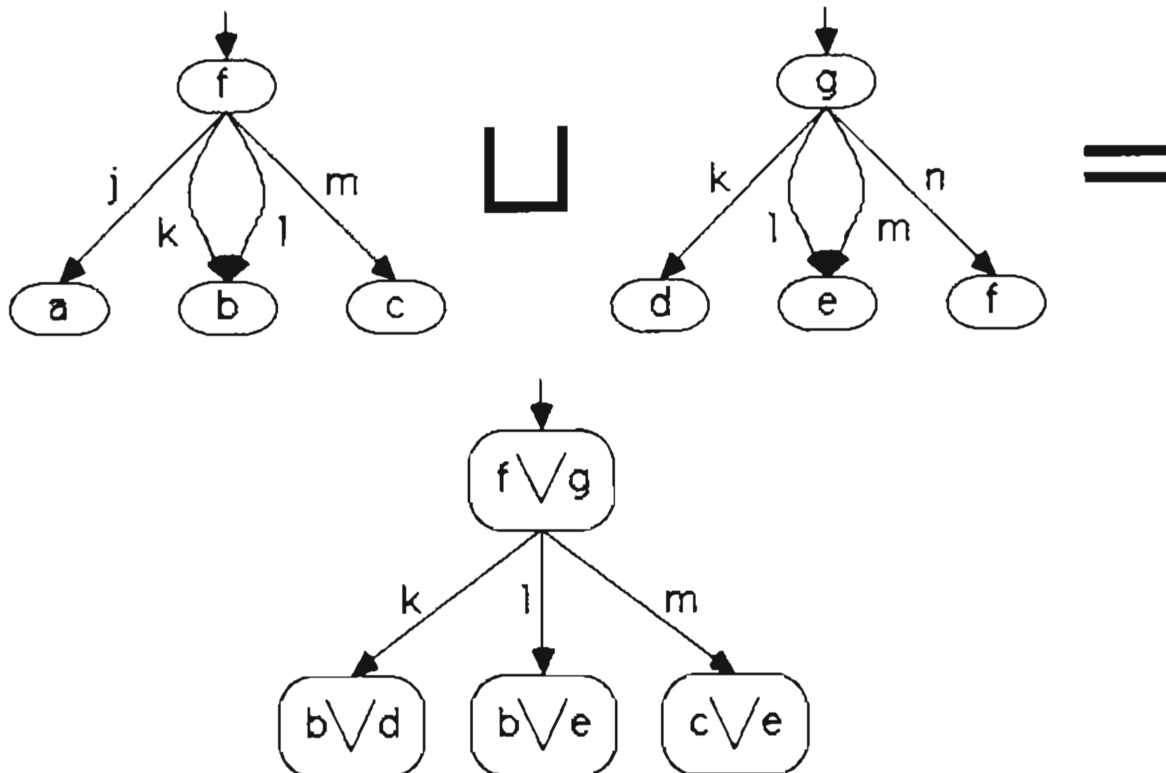

Figure 4.8: The join of two terms

For the unification of two terms t_1 and t_2, we want coreferring addresses in an argument to corefer in the result. Thus the coreference relation must (at least) contain the least equivalence relation on $\mathbf{A}$ containing both $\equiv_1$ and $\equiv_2$. Start by letting $\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$ be the set of addresses that occur in the arguments t_1 or t_2. Next, extend the equivalence relations $\equiv_1$ and $\equiv_2$ to $\mathbf{A}$ by letting $\equiv'_1$ and $\equiv'_2$ be their reflexive extensions onto $\mathbf{A}$. Since the coreference relation must also be transitive, we define $\equiv$ by composing $\equiv'_1$ and $\equiv'_2$ and taking transitive closure:

$$\equiv \; = \; (\equiv'_1 \cup \equiv'_2)+$$

Unfortunately, this definition is not good enough since, for example, a and b may be placed in the same equivalence class in $\equiv$ without a.c and b.c being placed in the same equivalence class. The problem is that $\equiv$ is not necessarily right-invariant.

Example 27. Consider the two terms

t_6 = `f(j => X: g(k=>U, l=>V), m => X)`
t_7 = `f(n => Y: g(k=>W), m => Y)`

In the coreference relation $\equiv_6$, the addresses `j` and `m` corefer and in $\equiv_7$ the addresses `n` and `m` corefer. By extending them to $\equiv_6'$ and $\equiv_7'$ and taking the transitive closure, `j` and `n` are made to corefer. However, `j.k` does not corefer with `n.k`. □

If there exist two addresses a, b $\in \mathbf{A}$ such that a $\equiv$ b, then what we want is, for any address a.c, to add b.c to the equivalence class of a.c so that a.c $\equiv$ b.c. To emphasize that the resulting equivalence is a function of $\equiv_1$ and $\equiv_2$, we define an infix operator **ext** with the following inductive definitions:

$$\equiv_1 \mathbf{ext}^0 \equiv_2 \; = \; (\equiv'_1 \cup \equiv'_2)+$$

$$\equiv_1 \mathbf{ext}^k \equiv_2 \; = \; (\equiv_1 \mathbf{ext}^{k-1} \equiv_2) \bigcup \{ \langle a.c, b.c\rangle \mid \langle a,b\rangle \in (\equiv_1 \mathbf{ext}^{k-1} \equiv_2), \; c \in \mathbf{F}, \text{ and either } a.c \in \mathbf{A} \text{ or } b.c \in \mathbf{A} \}$$

$$\equiv_1 \mathbf{ext} \equiv_2 \; = \; \bigcup_{k=0}^{\infty} (\equiv_1 \mathbf{ext}^{k-1} \equiv_2)$$

Is **ext** an equivalence relation? Examining the definition of **ext** for coreference relations, we see that if we add <a,b> at step k, we also add <b,a> at step k. If we add <a,b> at step k, <b,b> will be added by step k+1. (It is not sufficient to note that b **ext** b by step 0 since, as we will see in the example below, b might not have been included at all in step 0.) And if <a,b> and <b,c> are added at step k, <a,c> will also be added by step k. (To see that transitivity holds, let a = a′.a″, b = b′.b″, and c = c′.c″ be addresses where a″, b″, c″ ∈ **F**. First, note that transitivity holds in $\mathbf{ext}^0$. Next, assume that transitivity holds in step k, i.e., <a′,b′> and <b′,c′> implies <a′,c′>. If <a′.a″, b′.b″> and <b′.b″, c′.c″> are added at step k+1, then it must be because a″ = b″ and b″ = c″. Consequently, since a″ = c″, <a′.a″, c′.c″> will also be added at step k+1.) Consequently, $\equiv_1$ **ext** $\equiv_2$ is an equivalence relation. An induction argument on k shows that **ext** for coreference relations is also commutative and associative since union (∪) is commutative and associative.

Example 27 (continued). We begin by constructing the transitive closure $\equiv_6 \mathbf{ext}^0 \equiv_7$, in which `j` and `n` corefer. In the inductive construction of $\equiv_6$ **ext** $\equiv_7$, we see that since `j.1` is in **A** and since `j` and `n` corefer, `j.1` is made to corefer with `n.1`. However, `n.1` is not even in **A**! □

To remedy the anomaly shown in Example 27, we define $\mathbf{A}_1$ **ext** $\mathbf{A}_2$ to be the smallest set of addresses containing all the addresses in t_1 and t_2 and all the addresses generated in the construction of $\equiv_1$ **ext** $\equiv_2$. Formally, the definition of **ext** for address

domains is:

$$\mathbf{A}_1\ \mathbf{ext}^0\ \mathbf{A}_2 = \mathbf{A}_1 \cup \mathbf{A}_2$$

$$\mathbf{A}_1\ \mathbf{ext}^k\ \mathbf{A}_2 = (\mathbf{A}_1\ \mathbf{ext}^{k-1}\ \mathbf{A}_2) \cup \{\ b.c\ |\ a.c \in (\mathbf{A}_1\ \mathbf{ext}^{k-1}\ \mathbf{A}_2) \text{ and } <a,b> \in (\equiv_1\ \mathbf{ext}^{k-1} \equiv_2)\ \}$$

$$\mathbf{A}_1\ \mathbf{ext}\ \mathbf{A}_2 = \bigcup_{k=0}^{\infty} (\mathbf{A}_1\ \mathbf{ext}^k\ \mathbf{A}_2)$$

Note that the definition of $\mathbf{ext}^k$ for address domains depends on the definition of $\mathbf{ext}^{k-1}$ for coreference relations; the former essentially constructs all the addresses needed by the definition of **ext** for coreference relations. Also note that $\mathbf{A}_1$ **ext** $\mathbf{A}_2$ is finite if $\mathbf{A}_1$ and $\mathbf{A}_2$ are finite. Finally, since union ($\cup$) and **ext** over coreference relations are both commutative and associative, an inductive argument on k shows that **ext** over address domains is also commutative and associative.

Example 28. For t_6 and t_7 of Example 27,

$$\mathbf{A}_6 = \{\epsilon,\ \mathtt{j},\ \mathtt{j.k},\ \mathtt{j.l},\ \mathtt{m}\}$$
$$\mathbf{A}_7 = \{\epsilon,\ \mathtt{n},\ \mathtt{n.k},\ \mathtt{m}\}$$
$$\mathbf{A}_1\ \mathbf{ext}^0\ \mathbf{A}_2 = \{\epsilon,\ \mathtt{j},\ \mathtt{j.k},\ \mathtt{j.l},\ \mathtt{m},\ \mathtt{n},\ \mathtt{n.k}\}$$
$$\mathbf{A}_1\ \mathbf{ext}^1\ \mathbf{A}_2 = \mathbf{A}_1\ \mathbf{ext}\ \mathbf{A}_2 = \{\epsilon,\ \mathtt{j},\ \mathtt{j.k},\ \mathtt{j.l},\ \mathtt{m},\ \mathtt{n},\ \mathtt{n.k},\ \mathtt{n.l}\}$$

□

Lemma 1. Assume $\equiv_1$ is a coreference relation over $\mathbf{A}_1$ and that $\equiv_2$ is a coreference relation over $\mathbf{A}_2$ and that $\mathbf{A}_1 \subseteq \mathbf{A}_2$ and $\equiv_1 \subseteq \equiv_2$. Then $\mathbf{A}_1$ **ext** $\mathbf{A}_2 = \mathbf{A}_2$ and $\equiv_1$ **ext** $\equiv_2 = \equiv_2$. □

Proof. When $\equiv_1 \subseteq \equiv_2$ we see that $\equiv_1\ \mathbf{ext}^0 \equiv_2 = \equiv_2$. When $\mathbf{A}_1 \subseteq \mathbf{A}_2$ we see that $\mathbf{A}_1\ \mathbf{ext}^0\ \mathbf{A}_2 = \mathbf{A}_2$. Assuming $\equiv_1\ \mathbf{ext}^{k-1} \equiv_2 = \equiv_2$, we have $\equiv_1\ \mathbf{ext}^k \equiv_2 = \equiv_2$. Assuming $\equiv_1\ \mathbf{ext}^{k-1} \equiv_2 = \equiv_2$ and $\mathbf{A}_1\ \mathbf{ext}^{k-1}\ \mathbf{A}_2 = \mathbf{A}_2$, we have $\mathbf{A}_1\ \mathbf{ext}^k\ \mathbf{A}_2 = \mathbf{A}_2$, By induction on k, we get $\equiv_1$ **ext** $\equiv_2 = \equiv_2$ and $\mathbf{A}_1$ **ext** $\mathbf{A}_2 = \mathbf{A}_2$. □

Now that we have said what the address domain and coreference structure are to look like, we are ready to define the unification operator.

Definition. The *unification* of two terms $t_1 = <\mathbf{A}_1, \psi_1, \tau_1>$ and $t_2 = <\mathbf{A}_2, \psi_2, \tau_2>$ (denoted $t_1 \sqcap t_2$) is the term $t = <\mathbf{A}, \psi, \tau>$ where

(i) $\mathbf{A} = \mathbf{A}_1$ ext $\mathbf{A}_2$

(ii) τ is a function (in $\mathbf{A} \rightarrow \mathbf{V}$) such that kernel($\tau$) is $\equiv_1$ **ext** $\equiv_2$, and

(iii) $\psi(a) = \psi_1(b_1) \wedge ... \wedge \psi_1(b_m) \wedge \psi_2(c_1) \wedge ... \wedge \psi_2(c_n)$ for all $b_i \in \mathbf{A}_1$ such that $<a,b_i> \in (\equiv_1 \textbf{ ext } \equiv_2)$ and for all $c_i \in \mathbf{A}_2$ such that $<a,c_i> \in (\equiv_1 \textbf{ ext } \equiv_2)$, where $\wedge$ is the G.L.B. operation in the signature lattice. □

Part (iii) of this definition says approximately $\psi(a) = \psi_1(a) \wedge \psi_2(a)$ for all $a \in \mathbf{A}$. The more complex definition of ψ is needed since ψ_1 (or ψ_2) may not actually be defined for all a in **A**. But note that $<a,a> \in (\equiv_1 \textbf{ ext } \equiv_2)$ so that the definition of ψ given implies $\psi(a) = \psi_1(a) \wedge \psi_2(a)$ when ψ_1 and ψ_2 are defined on address a.

The definition of unification is well-formed since $t_1 \approx \bot$ implies $t_1 \sqcap t_2 \approx \bot$ and since alphabetic variation will not effect the result.

The definition of unification is more complex than the definition of generalization because it is easier to throw information away (which is effectively what generalization does) than it is to consistently integrate two sources of information, which is the task of unification.

Example 29. The unification of t_1 and t_2 from the previous two examples is:

```
f(j ⇒ X:g(k⇒U, l⇒V),
  m ⇒ X,
  n ⇒ X)
```

□

Example 30. When the ψ-term

```
(agreement ⇒ X:(number    ⇒ singular),
 subject   ⇒    (agreement ⇒ X))
```

is unified with

```
(subject ⇒ (agreement ⇒ (person ⇒ third)))
```

the result is

```
(agreement ⇒ X:(number ⇒ singular,
                person ⇒ third),
 subject   ⇒ (agreement ⇒ X))
```

This example, which appeared in Chapter 3, was adapted from the PATR literature [Schieber 1985a, page 19]. □

Example 31. Assume that a `properNP` is a kind of `np`, i.e., `properNP < np`, and that the intersection of `male` with `child` is `boy`, i.e., that `male∧child=boy`. Then the term

```
properNP(lex  ⇒ chris,
         type ⇒ child)
```

unifies with

```
np(lex         ⇒ X,
   translation ⇒ X,
   type        ⇒ male,
   role        ⇒ subject,
   number      ⇒ singular)
```

to give

```
properNP(lex         ⇒ X:chris,
         translation ⇒ X,
         type        ⇒ boy,
         role        ⇒ subject,
         number      ⇒ singular)
```

In this example, we see a hint of how the mechanism of unification can be used to

perform semantic type-class reasoning. □

Now for the promised result.

Theorem 4. Ψ is a lattice with respect to $\sqsubseteq$. The operators $\sqcup$ and $\sqcap$ are the join and meet operations. □

Proof. (original) It is sufficient to show the following:

(1a) $x \sqcap x = x$ (idempotency)
(1b) $x \sqcup x = x$ (idempotency)
(2a) $x \sqcap y = y \sqcap x$ (commutativity)
(2b) $x \sqcup y = y \sqcup x$ (commutativity)
(3a) $x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$ (associativity)
(3b) $x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z$ (associativity)
(4a) $x \sqcap (x \sqcup y) = x$ (absorption)
(4b) $x \sqcup (x \sqcap y) = x$ (absorption)

To show that $\sqsubseteq$ is the lattice order, we must also show:

(5) $x \sqsubseteq y$ is equivalent to $x \sqcap y = x$ and $x \sqcup y = y$ (consistency)

(There are other approaches to constructing a lattice of ψ-terms. We could, for example, have taken the ordering $\sqsubseteq$ and shown that for every term t_1 and t_2, there is a t_3 such that $t_3 \sqsubseteq t_1$ and $t_3 \sqsubseteq t_2$ and for for any t_4 such that $t_4 \sqsubseteq t_1$ and $t_4 \sqsubseteq t_2$, we have $t_4 \sqsubseteq t_3$. Finally, we would show our definition of $\sqcap$ is the same term as t_3. And symmetrically for $\sqcup$. Alternately, we could have shown that the term t_3 computed by $\sqcap$ as we defined it on ψ-terms satisfies the glb conditions, namely that $t_3 \sqsubseteq t_1$ and $t_3 \sqsubseteq t_2$ and for any t_4 such that $t_4 \sqsubseteq t_1$ and $t_4 \sqsubseteq t_2$, we have $t_4 \sqsubseteq t_3$. And symmetrically for $\sqcup$. We choose the approach of proving the lattice axioms because it seems easier.)

We use $\mathbf{A}_x$ to denote the address domain of term x and $\mathbf{A}_{x \sqcap y}$ to denote the address domain of term $x \sqcap y$, etc. Likewise, we use subscripts on $\equiv$ and ψ to indicate the relevant terms.

(1a) Idempotency of unification follows from the definition of $\sqcap$ since $\mathbf{A}_x =$

$(\mathbf{A}_x \ \mathbf{ext} \ \mathbf{A}_x)$ and $\equiv_x = (\equiv_x \ \mathbf{ext} \equiv_x)$.

(1b) Idempotency of generalization follows from the definition of $\sqcup$ since $\mathbf{A}_x = \mathbf{A}_x \cap \mathbf{A}_x$ and $\equiv_x = \equiv_x \cap \equiv_x$.

(2a) Unification ($\sqcap$) is commutative because it is defined from commutative operators; that is $(\mathbf{A}_x \ \mathbf{ext} \ \mathbf{A}_y) = (\mathbf{A}_y \ \mathbf{ext} \ \mathbf{A}_x)$, $(\equiv_x \ \mathbf{ext} \equiv_y) = (\equiv_y \ \mathbf{ext} \equiv_x)$, and $\psi_x(a) \wedge \psi_y(b) = \psi_y(b) \wedge \psi_x(a)$.

(2b) Commutativity of $\sqcup$ follows from the commutativity of the operators ($\cap$ and $\vee$) in terms of which $\sqcup$ is defined.

(3a) Associativity of $\sqcap$ follows because **ext** over addresses is associative, because **ext** over coreference relations is associative, and because $\wedge$ is associative.

(3b) Associativity of $\sqcup$ follows from the associativity of $\cap$ and $\vee$.

(4a) Let $\equiv_{x \sqcup y} = \equiv_x \cap \equiv_y$, $\mathbf{A}_{x \sqcup y} = \mathbf{A}_x \cap \mathbf{A}_y$, and $\psi_{x \sqcup y}(a) = \psi_x(a) \vee \psi_y(a)$ (for all a $\in \mathbf{A}_{x \sqcup y}$) describe $x \sqcup y$. Since $\equiv_{x \sqcup y} \subseteq \equiv_x$, we have $\equiv_{x \sqcup y} \ \mathbf{ext} \equiv_x = \equiv_x$ by Lemma 1. Since $\mathbf{A}_{x \sqcup y} \subseteq \mathbf{A}_x$, we have $\mathbf{A}_{x \sqcup y} \ \mathbf{ext} \ \mathbf{A}_x = \mathbf{A}_x$ by Lemma 1. Since Σ is a lattice, we have $\psi_{x \sqcup y}(a) \wedge \psi_x(a) = \psi_x(a)$. Thus, by the definition of $\sqcap$, we have $x \sqcap (x \sqcup y) = x$.

(4b) Let $\equiv_{x \sqcap y} = \equiv_x \ \mathbf{ext} \equiv_y$, $\mathbf{A}_{x \sqcap y} = \mathbf{A}_x \ \mathbf{ext} \ \mathbf{A}_y$, and $\psi_{x \sqcap y}(a) = \psi_x(b_i) \wedge \psi_y(c_j)$ (for all $a \equiv_{x \sqcap y} b_i$ and $a \equiv_{x \sqcap y} c_j$) describe $x \sqcap y$. Clearly $\equiv_x \ \mathbf{ext} \equiv_y \supseteq \equiv_x$ since the definition of **ext** only adds relationships to the constructed coreference relation. Likewise, $\mathbf{A}_x \ \mathbf{ext} \ \mathbf{A}_y \supseteq \mathbf{A}_x$ since the definition of **ext** for addresses only adds addresses to the constructed set. So $\equiv_{x \sqcap y} \supseteq \equiv_x$ and $\mathbf{A}_{x \sqcap y} \supseteq \mathbf{A}_x$. Since Σ is a lattice, $\psi_{x \sqcap y}(a) \leq \psi_x(a)$. Let $\equiv_{x \sqcup (x \sqcap y)} = \equiv_x \cap \equiv_{x \sqcap y}$, and let $\mathbf{A}_{x \sqcup (x \sqcap y)} = \mathbf{A}_x \cap \mathbf{A}_{x \sqcap y}$, and let $\psi_{x \sqcup (x \sqcap y)}(a) = \psi_x(a) \vee \psi_{x \sqcap y}(a)$, for every $a \in \mathbf{A}_{x \sqcup (x \sqcap y)}$. Since $\equiv_{x \sqcup (x \sqcap y)} \subseteq \equiv_x$, we have $\equiv_{x \sqcup (x \sqcap y)} = \equiv_x$ by Lemma 1. Since $\mathbf{A}_{x \sqcup (x \sqcap y)} \subseteq \mathbf{A}_x$, we have $\mathbf{A}_{x \sqcup (x \sqcap y)} = \mathbf{A}_x$ by Lemma 1. And since Σ is a lattice, we have $\psi_{x \sqcup (x \sqcap y)}(a) = \psi_x(a)$. Thus $x \sqcup (x \sqcap y) = x$.

So Ψ is a lattice with G.L.B. given by our definition of $\sqcap$ and L.U.B. given by our definition of $\sqcup$. For subsumption, we break the equivalence into 4 cases.

(5a) Assume $x \sqsubseteq y$. By the definition of $\sqsubseteq$ we have $\mathbf{A}_y \subseteq \mathbf{A}_x$ and $\equiv_y \subseteq \equiv_x$ and $\psi_x(a) \leq \psi_y(a)$. From Lemma 1, we conclude that $\mathbf{A}_x \ \mathbf{ext} \ \mathbf{A}_y = \mathbf{A}_x$ and $\equiv_x \ \mathbf{ext} \equiv_y = \equiv_x$. Since Σ is a lattice, $\psi_x(a) \leq \psi_y(a)$ implies $\psi_x(a) \wedge \psi_y(a) = \psi_x(a)$. Note that since $\equiv_{x \sqcap y} = \equiv_x$, $\psi_x(a) \wedge \psi_y(a) = \psi_{x \sqcap y}(a)$. Thus $x \sqcap y = x$.

(5b) Assume $x \sqsubseteq y$. Then $\mathbf{A}_y \subseteq \mathbf{A}_x$ implies $\mathbf{A}_x \cap \mathbf{A}_y = \mathbf{A}_y$, and $\equiv_y \subseteq \equiv_x$ implies $\equiv_x \cap \equiv_y = \equiv_y$, and $\psi_x(a) \leq \psi_y(a)$ implies $\psi_{x \sqcup y}(a) = \psi_x(a) \vee \psi_y(a) = \psi_y(a)$. Thus $x \sqcup y = y$.

(5c) Assume $x \sqcup y = y$. From $\mathbf{A}_x \cap \mathbf{A}_y = \mathbf{A}_y$ conclude $\mathbf{A}_y \subseteq \mathbf{A}_x$. From

$\equiv_x \cap \equiv_y = \equiv_y$ conclude $\equiv_y \subseteq \equiv_x$. From $\psi_x(a) \vee \psi_y(a) = \psi_y(a)$ conclude $\psi_x(a) \leq \psi_y(a)$. Thus $x \sqsubseteq y$.

(5d) Assume $x \sqcap y = x$. Since step 0 in the construction of $\mathbf{A}_x$ **ext** $\mathbf{A}_y$ begins by adding in all addresses in $\mathbf{A}_y$, from $\mathbf{A}_x$ **ext** $\mathbf{A}_y = \mathbf{A}_x$ conclude $\mathbf{A}_y \subseteq \mathbf{A}_x$. Likewise, since step 0 in the construction of $\equiv_x$ **ext** $\equiv_y$ begins by adding in all equivalences in $\equiv_y$, from $\equiv_x$ **ext** $\equiv_y = \equiv_x$ conclude $\equiv_y \subseteq \equiv_x$. From $\psi_x(a) \wedge \psi_y(b_i) = \psi_x(a)$ for all $a \in \mathbf{A}_x$ and all $b_i \in \mathbf{A}_y$ such that $a \equiv_{x \sqcap y} b_i$, conclude $\psi_x(b) \leq \psi_y(b)$ for all $b \in \mathbf{A}_y$ since $\mathbf{A}_y \subseteq \mathbf{A}$ and Σ is a lattice. Thus $x \sqsubseteq y$.

□

The top of the lattice is the most general term, $\top$, since $t \sqsubseteq \top$ for every term t. The bottom of the lattice is $\bot$ since $\bot \sqsubseteq t$ for every term t.

Theorem 5. Ψ is not a distributive lattice. □

Proof. (Ait-Kaci) We must show that

$$r \sqcap (s \sqcup t) = (r \sqcap s) \sqcup (r \sqcap t)$$

does not always hold. Let

r = f

s = f(1⇒a)

t = g

Assume that f, g, and a are unrelated in Σ. Then

f ⊓ (f(1⇒a) ⊔ g) = f

(f ⊓ f(1⇒a)) ⊔ (f ⊓ g) = f(1⇒a)

□

A traditional first-order term can be translated into a ψ-term by generating features for the arguments of the first-order term from the sequence 1, 2, 3, ... First-order variables are translated into corresponding variables from **V** and functor and constant symbols are translated into corresponding symbols in Σ.

Example 32. The first-order term

```
f(a, g(X), Y, X)
```

would be translated to:

```
f(1 ⇒ a,
  2 ⇒ g(1⇒X),
  3 ⇒ Y,
  4 ⇒ X)
```

□

When the signature is flat, i.e., when unequal symbols are unrelated, unification of ψ-terms so generated is very similar to unification of first-order terms.

Example 33. Unifying the first-order term from the previous example with

```
f(a, g(b), Z, Z)
```

gives

```
f(a, g(b), b, b)
```

Unifying the ψ-term in that example with

```
f(1 ⇒ a,
  2 ⇒ g(1⇒b),
  3 ⇒ Z,
  4 ⇒ Z)
```

gives

```
f(1 ⇒ a,
  2 ⇒ g(1⇒b),
  3 ⇒ b,
  4 ⇒ b)
```

□

Let h represent the first-order to ψ-term translation and let $\sqcap_{f.o.}$ represent first-order unification. Unfortunately, the following statement is not quite true:

$$h(t_1) \sqcap h(t_2) = h(\ t_1 \sqcap_{f.o.} t_2\ )$$

One difference from first-order unification occurs when the number of attributes/arguments varies.

Example 34. The first-order terms `f(a)` and `f(X,b)` do not unify. But when translated to ψ-terms they do. The result is the ψ-term translation of `f(a,b)`. □

We can remedy this problem by adding another attribute, called `arity`, to every ψ-term so, for example, the first-order term `f(X,b)` would be translated to the following ψ-term:

```
f(arity ⇒ 2,
  1     ⇒ X,
  2     ⇒ b)
```

In some discussions of first-order logic, two functors with the same spelling are considered to be distinct if they are used with different numbers of arguments. To emphasize this difference, a functor such as `f` above is renamed `f/2` to indicate that it takes two arguments.

Another difference between first-order and ψ-logic has to do with the way address coreference is dealt with.

Example 35. Let

$$t_1 = \texttt{f(a,a)}$$

and

$$t_2 = \texttt{f(X,X)}$$

These two first-order terms unify to produce

$$t_1 \sqcap t_2 = \texttt{f(a,a)}$$

The two ψ-term translations of t_1 and t_2

```
h(t1) = f(arity ⇒ 2,
          1     ⇒ a,
          2     ⇒ a)
h(t2) = f(arity ⇒ 2,
          1     ⇒ X,
          2     ⇒ X)
```

unify to produce the ψ-term

```
h(t2) ⊓ h(t2) = f(arity ⇒ 2,
                  1     ⇒ X:a,
                  2     ⇒ X)
```

That term is not equal to the translation of the first-order unification

```
h(t2 ⊓ t2) = f(arity ⇒ 2,
               1     ⇒ a,
               2     ⇒ a)
```

When the result of a first-order unification violates the *occurs-check*, producing an infinite term, the result is declared invalid and the two first-order terms are said not to unify. Their ψ-term translations do, however, unify producing a cyclic ψ-term as the result.

4.4. Extending Horn Clause Logic to ψ-Logic

The next few definitions describe ψ-programs. Intuitively, a ψ-program is just like a first-order logic program except ψ-terms are used instead of literals and terms. An Inheritance Grammar is just a ψ-program. We begin with the definition of ψ-clauses, which will be revised later on.

Definition (first cut). A *ψ-clause* is a sequence of k+1 ψ-terms ($k \geq 0$) and is written as:

t_0 :- $t_1, t_2, \ldots, t_k$.

or as

t_0.

when $k = 0$. The first term, t_0, is called the *positive* ψ-literal or the *clause head* and $t_1, t_2, \ldots, t_k$ are the *negative* ψ-literals or the *body*. We also call ψ-clauses ψ-*rules*. □

Just as in Horn clauses, the scope of variables names is the entire clause. Analogously to well-formed terms, a ψ-clause t_0 :- $t_1, t_2, \ldots, t_k$ is *well-formed* if the same subterm occurs at all addresses that have the same variable.

To make the definition of ψ-clauses more rigorous, we must extend the concept of address coreference to ψ-clauses and then stipulate that two addresses a_i and a_j that corefer (possibly from distinct t_i and t_j) must reference the same subterm. We begin by naming the literals in a clause with integer indices. For example, the literals in the clause

$$t_0 \text{ :- } t_1, t_2, \ldots, t_k.$$

are named from the index set 0, 1, ... , k. Then, we prepend these indices to addresses so that subterms can properly be located within an entire clause.

Definition. Given a set of ψ-terms $t_0, \ldots, t_k$, define the *shared address domain*, **A**, as

$$\mathbf{A} = \{ \text{ i.a } \mid a \in \mathbf{A}_i \text{ for } 0 \leq i \leq k \}$$

Technically, we must add the indices 0, ... , k to **F** if not already present. □

Example 36. Consider the clause

```
f(m⇒a, n⇒X) :-
    g(m⇒a, n⇒h(p⇒c, q⇒X)).
```

with the head labeled 0 and the single body term labeled 1. The shared address domain used in this clause is

{ 0⇒, 0⇒m⇒, 0⇒n⇒, 1⇒, 1⇒m⇒,
1⇒n⇒, 1⇒n⇒p⇒, 1⇒n⇒q⇒ }.

Note that a shared address domain has no root address. □

Although we could have easily defined shared address domains independently of ψ-clauses (just as we defined address domains independently of ψ-terms), we chose this approach to emphasize the connection between the shared address domain and the address domains $\mathbf{A}_i$ of which it is constructed.

Since variables in a ψ-clause are to be quantified over the entire clause, we must construct an equivalence relation $\equiv$ over the shared address domain.

Definition. Let τ_i be the variable tagging functions for the terms in a set of ψ-terms $t_0, \ldots, t_k$ and let $\mathbf{A}$ be the shared address domain of the set of ψ-terms. The *shared variable tagging function*

$$\tau: \mathbf{A} \rightarrow \mathbf{V}$$

is defined as

$$\tau(i.a) = \tau_i(a) \qquad \text{for all } a \in \mathbf{A}_i$$

The kernel of τ induces a *shared address coreference relation*, denoted $\equiv$. □

Example 37. In the shared address domain $\equiv$ of Example 36, the addresses 0⇒n⇒ and 1⇒n⇒q⇒ corefer. □

Similarly, we extend the symbol mapping functions, ψ_i.

Definition. Let ψ_i be the symbol mapping functions for the terms in the set of ψ-terms $t_0, \ldots, t_k$. The *shared symbol mapping*

$$\psi: \mathbf{F}^* \rightarrow \Sigma$$

is defined as

$$\psi(i.a) = \psi_i(a) \qquad \text{for all } a \in A_i$$
$$\psi(X) = \top \qquad \text{otherwise}$$

□

Definition. A *set of shared ψ-terms*, T, is a quadruple $<\mathbf{S}, \mathbf{A}, \psi, \tau>$ where

S is an index set, i.e., 0, 1, ... k,

A is a shared address domain,

ψ is a shared symbol mapping, $F^* \rightarrow \Sigma$, and

τ is a shared variable tagging, $\mathbf{A} \rightarrow \mathbf{V}$.

□

A ψ-clause is just a set of shared ψ-terms with a distinguished head term.

Definition (final). A *ψ-clause* is a quintuple $<\mathbf{S}, s, \mathbf{A}, \psi, \tau>$ where $<\mathbf{S}, \mathbf{A}, \psi, \tau>$ is a set of shared ψ-terms and $s \in \mathbf{S}$ is called the *head index.* □

We assume that clauses are indexed by 0, 1, ... , k with head 0 unless explicitly stated otherwise.

Example 38. Let C_1 be the quintuple $<\mathbf{S}, s, \mathbf{A}, \psi, \tau>$ where

S = { 0, 1, 2 },

s = 0, and

A = { 0⇒, 0⇒i⇒, 0⇒j⇒, 0⇒j⇒k⇒,

1⇒, 1⇒l⇒, 1⇒m⇒, 1⇒m⇒n⇒, 2⇒ }

and where ψ includes

$$
\begin{aligned}
&\psi(\texttt{0}\Rightarrow) = \texttt{a}\\
&\psi(\texttt{0}\Rightarrow\texttt{i}\Rightarrow) = \texttt{b}\\
&\psi(\texttt{0}\Rightarrow\texttt{j}\Rightarrow) = \texttt{c}\\
&\psi(\texttt{0}\Rightarrow\texttt{j}\Rightarrow\texttt{k}\Rightarrow) = \texttt{d}\\
&\psi(\texttt{1}\Rightarrow) = \texttt{e}\\
&\psi(\texttt{1}\Rightarrow\texttt{l}\Rightarrow) = \texttt{c}\\
&\psi(\texttt{1}\Rightarrow\texttt{l}\Rightarrow\texttt{k}\Rightarrow) = \texttt{d}\\
&\psi(\texttt{1}\Rightarrow\texttt{m}\Rightarrow) = \texttt{f}\\
&\psi(\texttt{1}\Rightarrow\texttt{m}\Rightarrow\texttt{n}\Rightarrow) = \texttt{g}\\
&\psi(\texttt{2}\Rightarrow) = \texttt{f}\\
&\psi(\texttt{2}\Rightarrow\texttt{n}\Rightarrow) = \texttt{g}
\end{aligned}
$$

(all other values of ψ are $\top$) and where τ includes

$$
\begin{aligned}
&\tau(\texttt{0}\Rightarrow\texttt{j}\Rightarrow) = \texttt{X}\\
&\tau(\texttt{1}\Rightarrow\texttt{l}\Rightarrow) = \texttt{X}\\
&\tau(\texttt{1}\Rightarrow\texttt{m}\Rightarrow) = \texttt{Y}\\
&\tau(\texttt{2}\Rightarrow) = \texttt{Y}
\end{aligned}
$$

(all other values of τ are distinct). C_1 is a clause. □

We extend the conventions for textual and graphical representation of ψ-terms to ψ-clauses. To associate a name (e.g., C_1) with a clauses, we will prefix the clause with its name followed by a colon.

Example 39. The textual representation of clause C_1 from the previous example is more enlightening:

```
a(i=>b, j=>X:c(k=>d)) :-
                e(l=>X:c(k=>d), m=>Y:f(n=>g)),
                Y:f(n=>g).
```

In its graphical representation in Figure 4.9, note the convention used to identify the head (index `0`) and body ψ-terms (indices `1`, `2`, ...). □

Definition. A set of shared ψ-terms

$$T: \quad t_0, t_1, \ldots, t_k.$$

is *well-formed* if the subterms occurring at coreferring addresses in T are equal. More specifically, if $a \equiv b$ then, for all $c \in \mathbf{F}^*$ such that $a.c \in \mathbf{A}$

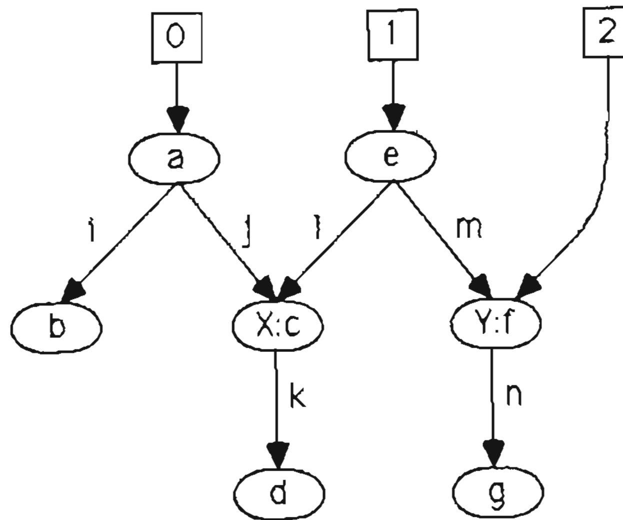

Figure 4.9: Graphical representation of C_1

(i) $b.c \in \mathbf{A}$,

(ii) $\psi(b.c) = \psi(a.c)$, and

(iii) $\tau(b.c) = \tau(a.c)$.

A ψ-clause is well-formed if the underlying set of shared ψ-terms is well-formed. □

Example 40. The ψ-clause C_1 from the previous example is well-formed. □

Clauses consist of terms and subterms. The ψ-term in clause C at address a, denoted C\a, is defined analogously to subterms of ψ-terms, i.e., as a restriction of $\mathbf{A}$, ψ, and τ. The term at address s is the clause head and the terms at the other indices in $\mathbf{S}$ comprise the body. For well-formed terms, $\mathbf{A}_i = \mathbf{A}\backslash i$, $\psi_i = \psi\backslash i$, and $\tau_i = \tau\backslash i$ hold.

Definition. For any well-formed clause, the terms $t_i = \langle \mathbf{A}_i, \psi_i, \tau_i \rangle$ (for $i \in \mathbf{S}$) are the *ψ-literals* from which the clause is constructed. □

Just as in ψ-terms, we ignore consistent variable renamings within a ψ-clause and treat all alphabetic variants as equal. Also, any clause containing the symbol $\perp$ at any address should be considered an error. Such error clauses will not arise during computation, so we will say little more about them. In particular, we won't bother to equate error clauses, since we are not constructing a lattice of clauses. Note that the index numbering is lost in the textual representation of a clause. It is generally irrelevant and we ignore the indices but, technically, two clauses identical up to indexing on their bodies are nevertheless distinct.

Recall that ψ-terms were defined in a context of fixed Σ, $\mathbf{F}$, and $\mathbf{V}$. Given the textual representation of several clauses, we can easily see which symbols, features, and variables are used so this is no problem. To specify the ordering of the symbols, we introduce *ordering statements*.

Definition. An *ordering statement* is a syntactic entity of the form:

$$\{s_1,\ s_2,\ \ldots,\ s_k\} < s_0.$$

for $k \geq 1$. When $k = 1$, we may also write

$$s_1 < s_0.$$

Each s_i is an identifier beginning with a lowercase letter or number. Within each ordering statement, we require $s_i \neq s_0$ for $1 \leq i \leq k$. □

Definition. Given a set of ordering statements, we can construct a *corresponding order* (represented with $\leq$) by assuming reflexivity and transitive closure. A set of ordering statements is *well-formed* if and only if the corresponding order is partial, i.e., if antisymmetry ($s_1 \leq s_2$ and $s_2 \leq s_1$ implies $s_1 = s_2$) also holds. □

Example 41. The following ordering statements are well-formed.

```
properNoun < noun.
{noun, verb} < word.
```

The corresponding order includes properNoun≤word and noun≤noun. □

Example 42. The following ordering statements are not well-formed.

```
a < b.
b < c.
c < a.
```

Since a≤c and c≤a but a≠c. □

Theorem 6. Any partial order (S_P, $\leq_P$) can be *embedded in* (or *extended to*) a lattice (S_L, $\leq_L$) that respects the partial order. That is $S_P \subseteq S_L$ and for any x, y $\in S_P$, $x \leq_L y$ if and only if $x \leq_P y$. □

Proof. (Birkhoff) Omitted; see [Maier 1980]. □

In general, there are many extensions, none necessarily minimal. We prefer an extension with fewer elements but, otherwise, any will do. We are now ready to describe ψ-programs.

Definition. A *ψ-program*, G, is a set of well-formed ψ-clauses and well-formed ordering statements. Σ is taken to comprise the symbols appearing in the ψ-clauses and ordering statements (and $\top$ and $\bot$). The lattice structure is taken to be an extension of the order corresponding to ordering statements. □

A ψ-program is also called a ψ-database or an *Inheritance Grammar*. Since the scope of variables is the clause and not the program, there is no concept of well-formed ψ-program.

A ψ-query is just a ψ-clause with no head, as given in the next definition.

Definition. A *ψ-query*, represented textually as

:- t_1, ... , t_k.

where $k \geq 0$, is a set of shared ψ-terms <**S**, **A**, ψ, τ>. A well-formed ψ-query is a ψ-query in which the same sub-term occurs at all addresses that are tagged with the same variable. Henceforth, we shall assume all ψ-queries are well-formed. □

4.5. A Resolution Rule for ψ-Clauses

In this section, we define a resolution rule for ψ-clauses. We view ψ-clause resolution as an operation that takes two ψ-clauses as arguments and produces a ψ-clause as a result. Just as in first-order resolution, a literal in the body of one clause C_1 is unified with the head of the second clause C_2. Consider the two clauses

$$C_1\text{:}\ t_0 \text{:-}\ t_1,\ \ldots\ ,\ t_j.$$
$$C_2\text{:}\ t_{j+1} \text{:-}\ t_{j+2},\ \ldots\ ,\ t_k.$$

where $0<j<k$. These clauses will unify exactly when some literal t_i ($1 \leq i \leq j$) in C_1 unifies with the head of C_2, i.e., $t_i \sqcap t_{j+1} \neq \perp$. The result is *approximately:*

$$C\text{:}\ t_0 \text{:-}\ t_0,\ \ldots\ ,\ t_{i-1},\ t_{i+1},\ \ldots\ ,\ t_j,\ t_{j+2},\ \ldots\ ,\ t_k.$$

This representation of the result is not quite correct for two reasons. First, it fails to take into account changes (i.e., substitutions) made as a result of the unification. Second, we need a little more precision in our manipulation of indices in constructing the result.

In the usual definition of first-order resolution, the unification operation results in a substitution which is then applied to the head and body literals of the result. However, ψ-unification produces a new ψ-term, not a substitution. It would certainly have been possible to define ψ-unification so that it produced a *substitution object* and to define how a substitution can be applied to a term or clause. But a first-order substitution is a very syntactic entity (built from the operation of textual substitution) while a ψ-substitution would be a more complex object, essentially containing all the

information in the two argument terms. Consequently, the approach we adopt instead is to define unification directly on ψ-clauses or, more accurately, on sets of shared ψ-terms.

Before progressing, note that the definitions of **ext** apply, as given above, to *shared* address domains and *shared* coreference relations. For convenience, these definitions are repeated here:

$$\mathbf{A} = \mathbf{A}_1 \cup \mathbf{A}_2$$

$$\equiv'_1 = \text{the reflexive extension of } \equiv_1 \text{ onto } \mathbf{A}.$$

$$\equiv'_2 = \text{the reflexive extension of } \equiv_2 \text{ onto } \mathbf{A}.$$

$$\equiv_1 \mathbf{ext}^0 \equiv_2 \;=\; (\equiv'_1 \circ \equiv'_2)+$$

$$\equiv_1 \mathbf{ext}^k \equiv_2 \;=\; (\equiv_1 \mathbf{ext}^{k-1} \equiv_2) \bigcup \{ \langle a.c,\, b.c \rangle \mid \langle a,b \rangle \in (\equiv_1 \mathbf{ext}^{k-1} \equiv_2) \text{ and either } a.c \in \mathbf{A} \text{ or } b.c \in \mathbf{A} \}$$

$$\equiv_1 \mathbf{ext} \equiv_2 \;=\; \bigcup_{k=0}^{\infty} (\equiv_1 \mathbf{ext}^{k-1} \equiv_2)$$

$$\mathbf{A}_1 \;\mathrm{ext}^0\; \mathbf{A}_2 = \mathbf{A}_1 \cup \mathbf{A}_2$$

$$\mathbf{A}_1 \;\mathbf{ext}^k\; \mathbf{A}_2 = (\mathbf{A}_1 \;\mathbf{ext}^{k-1}\; \mathbf{A}_2) \cup \{ b.c \mid a.c \in (\mathbf{A}_1 \;\mathbf{ext}^{k-1}\; \mathbf{A}_2) \text{ and } \langle a,b \rangle \in (\equiv_1 \mathbf{ext}^{k-1} \equiv_2) \}$$

$$\mathbf{A}_1 \;\mathrm{ext}\; \mathbf{A}_2 = \bigcup_{k=0}^{\infty} (\mathbf{A}_1 \;\mathrm{ext}^k\; \mathbf{A}_2)$$

Here is the definition of the unification of two sets of shared ψ-terms. Notice how the relationship between S_1 and S_2 affects the character of this operation: since all addresses begin with an index, the intersection of S_1 and S_2 determines which literals are "unified together".

Definition. Given two sets of shared ψ-terms

$$T_1 = \langle \mathbf{S}_1, \mathbf{A}_1, \psi_1, \tau_1 \rangle$$

$$T_2 = \langle \mathbf{S}_2, \mathbf{A}_2, \psi_2, \tau_2 \rangle$$

define their unification $T_1 \sqcap T_2$ as

$$T = \langle \mathbf{S}, \mathbf{A}, \psi, \tau \rangle$$

where

(i) $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$

(ii) $\mathbf{A} = \mathbf{A}_1 \ \mathbf{ext} \ \mathbf{A}_2$

(iii) τ is a function (in $\mathbf{A} \rightarrow \mathbf{V}$) such that kernel($\tau$) is $\equiv_1 \mathbf{ext} \equiv_2$, and

(iv) $\psi(a) = \psi_1(b_1) \wedge \ldots \wedge \psi_1(b_m) \wedge \psi_2(c_1) \wedge \ldots \wedge \psi_2(c_n)$ for all $b_i \in \mathbf{A}_1$ such that $\langle a,b_i \rangle \in (\equiv_1 \mathbf{ext} \equiv_2)$ and for all $c_i \in \mathbf{A}_2$ such that $\langle a,c_i \rangle \in (\equiv_1 \mathbf{ext} \equiv_2)$, where $\wedge$ is the G.L.B. operation in the signature lattice. □

Example 43. Let T_1 be the set of shared ψ-terms

$$\texttt{a}(f_1 \Rightarrow \texttt{c},\ f_2 \Rightarrow \texttt{X:d}(f_5 \Rightarrow \texttt{e},\ f_9 \Rightarrow \texttt{n})),$$

$$\texttt{b}(f_3 \Rightarrow \texttt{X},\ f_4 \Rightarrow \texttt{l}(f_6 \Rightarrow \texttt{m}))$$

with the terms indexed by `0` and `1`, respectively. Let T_2 be the set

$$\texttt{f}(f_3 \Rightarrow \texttt{Y:g}(f_5 \Rightarrow \texttt{Z:i}),\ f_4 \Rightarrow \texttt{h}(f_8 \Rightarrow \texttt{Z})),$$

$$\texttt{j}(f_7 \Rightarrow \texttt{Y},\ f_8 \Rightarrow \texttt{k})$$

indexed by `1` and `2`, respectively, so the `b(...)` term will be unified with the `f(...)` term. We have

$$A_1 = \{\ 0{\Rightarrow},\ 0{\Rightarrow}f_1{\Rightarrow},\ 0{\Rightarrow}f_2{\Rightarrow},\ 0{\Rightarrow}f_2{\Rightarrow}f_5{\Rightarrow},\ 0{\Rightarrow}f_2{\Rightarrow}f_9{\Rightarrow},\ 1{\Rightarrow},\ 1{\Rightarrow}f_3{\Rightarrow},\ 1{\Rightarrow}f_3{\Rightarrow}f_5{\Rightarrow},\ 1{\Rightarrow}f_3{\Rightarrow}f_9{\Rightarrow},\ 1{\Rightarrow}f_4{\Rightarrow},\ 1{\Rightarrow}f_4{\Rightarrow}f_6{\Rightarrow}\ \}$$

where

$$0 \Rightarrow f_2 \Rightarrow \ \equiv \ 1 \Rightarrow f_3 \Rightarrow$$
$$0 \Rightarrow f_2 \Rightarrow f_5 \Rightarrow \ \equiv \ 1 \Rightarrow f_3 \Rightarrow f_5 \Rightarrow$$
$$0 \Rightarrow f_2 \Rightarrow f_9 \Rightarrow \ \equiv \ 1 \Rightarrow f_3 \Rightarrow f_9 \Rightarrow$$

and

$$A_2 = \{ 1 \Rightarrow, \ 1 \Rightarrow f_3 \Rightarrow, \ 1 \Rightarrow f_3 \Rightarrow f_5 \Rightarrow, \ 1 \Rightarrow f_4 \Rightarrow, \ 1 \Rightarrow f_4 \Rightarrow f_6 \Rightarrow, \ 2 \Rightarrow, \ 2 \Rightarrow f_7 \Rightarrow, \ 2 \Rightarrow f_7 \Rightarrow f_5 \Rightarrow, \ 2 \Rightarrow f_8 \Rightarrow \}$$

where

$$1 \Rightarrow f_3 \Rightarrow \ \equiv \ 2 \Rightarrow f_7 \Rightarrow$$
$$1 \Rightarrow f_3 \Rightarrow f_5 \Rightarrow \ \equiv \ 1 \Rightarrow f_4 \Rightarrow f_6 \Rightarrow \ \equiv \ 2 \Rightarrow f_7 \Rightarrow f_5 \Rightarrow$$

The construction of **A** introduces the address

$$2 \Rightarrow f_7 \Rightarrow f_9 \Rightarrow$$

The construction of $\equiv$ introduces

$$0 \Rightarrow f_2 \Rightarrow f_9 \Rightarrow \ \equiv \ 1 \Rightarrow f_3 \Rightarrow f_9 \Rightarrow \ \equiv \ 2 \Rightarrow f_7 \Rightarrow f_9 \Rightarrow$$

The result of the unification is the set

$$a(f_1 \Rightarrow c, \ f_2 \Rightarrow X{:}dg(f_5 \Rightarrow Y{:}eim, \ f_9 \Rightarrow n)),$$

$$bf(f_3 \Rightarrow X, \ f_4 \Rightarrow hl(f_6 \Rightarrow Y)),$$

$$j(f_7 \Rightarrow X, \ f_8 \Rightarrow k)$$

indexed by 1, 2, and 2, respectively, where

$$dg = d \wedge g$$
$$eim = e \wedge i \wedge m$$
$$bf = b \wedge f$$
$$hl = h \wedge l$$

in the signature. □

Since the definition of ψ-clause unification deals with variable coreference using equivalence relations and not with specific variable mapping functions, variable names are unimportant. In other words, one variable may appear in two clauses without

causing confusion: clauses do not share variables. When S_1 and S_2 are disjoint, unification of two sets of shared ψ-terms simply concatenates the two sets. When $S_1 = S_2$, the definition performs pairwise unification of the ψ-terms, much the same way that the definition of ψ-term unification performs pairwise unification of the subterms of matching features.

Later, we will find it convenient to also have a subsumption ordering on sets of shared ψ-terms and on ψ-clauses. We provide this ordering relation in the next definition before going on.

Definition. A set of shared ψ-terms

$$T_1 = \langle \mathbf{S}_1, \mathbf{A}_1, \psi_1, \tau_1 \rangle$$

is subsumed by another set of shared ψ-terms

$$T_2 = \langle \mathbf{S}_2, \mathbf{A}_2, \psi_2, \tau_2 \rangle$$

written $T_1 \sqsubseteq T_2$ if

(i) $\mathbf{A}_2 \subseteq \mathbf{A}_1$ (Every address in T_2 is in T_1),

(ii) $\equiv_2 \subseteq \equiv_1$ (All addresses that corefer in T_2 also corefer in T_1; i.e., T_1 is more constrained), and

(iii) $\psi_1(a) \leq \psi_2(a)$ for all addresses $a \in \mathbf{A}_2$, where $\leq$ is the partial order on symbols (the symbol at any address in T_1 is less than the symbol at the corresponding address in T_2).

We say that a ψ-clause

$$C_1 = \langle \mathbf{S}_1, s_1, \mathbf{A}_1, \psi_1, \tau_1 \rangle$$

is subsumed by another ψ-clause

$$C_2 = \langle \mathbf{S}_2, s_2, \mathbf{A}_2, \psi_2, \tau_2 \rangle$$

if

$$<\mathbf{S}_1, \mathbf{A}_1, \psi_1, \tau_1> \sqsubseteq <\mathbf{S}_2, \mathbf{A}_2, \psi_2, \tau_2>$$

and $s_1 = s_2$. □

Before we can extend the definition of unification of sets of shared ψ-terms into the desired resolution rule, we need an operator to re-index terms in a clause. To perform the re-indexing, we introduce a post-fix *re-indexing operator*, using the following notation, where T is a set of k shared ψ-terms:

$$T \{ i_0 \leftarrow j_0, \dots , i_k \leftarrow j_k \}$$

The j's are the indices of T before re-indexing and the i's are the new indices for the corresponding terms after the re-indexing.

Example 44. Consider the set of shared ψ-terms

$$T: \quad t_0,\ t_1,\ t_2,\ t_3$$

with index set $S = \{0,\ 1,\ 2,\ 3\}$. We can re-index T with the expression

$$T \{ 41 \leftarrow 0,\ 42 \leftarrow 1,\ 43 \leftarrow 2,\ 44 \leftarrow 3 \}$$

and write the result as

$$T: \quad t_{41},\ t_{42},\ t_{43},\ t_{44}$$

where

$$\begin{aligned} t_{41} &= t_0 \\ t_{42} &= t_1 \\ t_{43} &= t_2 \\ t_{44} &= t_3 \end{aligned}$$

□

We also use this operator to remove a term from a shared ψ-term set by suitably restricting S, **A**, ψ, and τ. The symbol • is used to indicate which term(s) to remove.

Example 45. Using the terms from the previous example, with S = {0, 1, 2, 3}, the expression

$$T\{\ 41\leftarrow 0,\ \bullet\leftarrow 1,\ 42\leftarrow 2,\ \bullet\leftarrow 3\ \}$$

is the set of shared ψ-terms

$$t_{41},\ t_{42}$$

where $t_{41}=t_0$ and $t_{42}=t_2$. □

We are now ready to define ψ-clause resolution.

Definition. Consider two ψ-clauses

$$C_1 = <\mathbf{S}_1, s_1, \mathbf{A}_1, \psi_1, \tau_1>$$

$$C_2 = <\mathbf{S}_2, s_2, \mathbf{A}_2, \psi_2, \tau_2>$$

where

$$S_1 = \{0, \dots, j\}$$
$$S_2 = \{j+1, \dots, k\}$$
$$s_1 = 0$$
$$s_2 = j+1$$

for $0<j<k$, i.e.,

$$C_1:\quad t_0 \text{ :- } t_1, \ \dots\ , t_i, \ \dots\ , t_j.$$
$$C_2:\quad t_{j+1} \text{ :- } t_{j+2}, \ \dots\ , t_k.$$

Assume that, for some i $(0<i\leq j)$, $t_i \sqcap t_{j+1} \neq \perp$. C_1 and C_2 can be *resolved* to produce the *resolvent* clause

$$C = <\mathbf{S}, s, \mathbf{A}, \psi, \tau>$$

where $<\mathbf{S}, \mathbf{A}, \psi, \tau>$ is the shared term set

$$(\ C_1 \sqcap (\ C_2\{\ i\leftarrow j+1,\ j+2\leftarrow j+2,\ \dots\ ,\ k\leftarrow k\ \}\ )\ )$$
$$\{\ 0\leftarrow 0,\ \dots\ ,\ i\text{-}1\leftarrow i\text{-}1,\ \bullet\leftarrow i,\ i+1\leftarrow i+1,\ \dots\ ,\ j\leftarrow j\ ,\ j+2\leftarrow j+2,\ \dots\ ,\ k\leftarrow k\ \}$$

and where s=0, i.e., the resulting clause has the structure indicated approximately by

$$C: \; t_0 :- t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_j, t_{j+2}, \ldots, t_k.$$

□

This definition first re-indexes C_2 so that its head has the same index as term t_i in the body of C_1. Thus, when the unification is done, these two terms will be unified together and, in general, the other terms will be modified as a side-effect. Finally, we remove the unified term t_i from the result.

Technically, we have just defined resolution only on terms with disjoint index sets. Since, as mentioned earlier, the actual indices are unimportant, this definition is extended by allowing re-indexing of C_1 and C_2 as necessary to ensure $S_1 \cap S_2 = \emptyset$.

Example 46. Using the sets of shared ψ-terms from Example 43, we see that the clauses

```
a(f1 => c, f2 => X:d(f5 => e, f9 => n)) :-
        b(f3 => X, f4 => l(f6 => m)).
```

and

```
f(f3 => Y:g(f5 => Z:i), f4 => h(f6 => Z)) :-
        j(f7 => Y, f8 => k).
```

resolve to yield the clause

```
a(f1 => c, f2 => X:dg(f5 => eim, f9 => n)) :-
    j(f7 => X, f8 => k)
```

assuming that

$$\begin{aligned} &d \wedge g = dg \neq \bot \\ &e \wedge i \wedge m = eim \neq \bot \\ &b \wedge f \neq \bot \\ &h \wedge l \neq \bot \end{aligned}$$

in the signature. □

We can extend the resolution rule to resolve a ψ-query with a ψ-clause by ignoring term t_0 in the above definition.

Example 47. Let C_1 be the ψ-query

```
:- p, q, r.
```

and C_2 be the ψ-clause

```
q :- s, t, u.
```

Then C_1 and C_2 can be resolved to produce the ψ-query

```
:- p, r, s, t, u.
```

□

4.6. A Model-Theoretic Semantics for ψ-Programs

The definitions and development of a semantics presented in this section parallel the development done for traditional first-order logic. In giving a meaning to ψ-terms and ψ-clauses we must first describe a domain (the "real world"). After doing so, we define the meaning of ψ-clauses in terms of this domain by defining an interpretation. We then say what it means for an interpretation to satisfy a ψ-clause. A satisfying interpretation for a set of ψ-clauses is called a *model.*

We use ψ-terms for reasoning about the "real world" or a "domain of discourse." The only structure that we impose on the domain is that it consists of a (universal) set of objects, U, and a set of binary relations, **R**, amongst the elements of U. This approach should be contrasted to a semantics for traditional first-order logic where n-

ary functions are used[3] and where functions that map into **true/false** (i.e., relations) are segregated from the other functions. We also stipulate that the universe **U** must contain at least one element, which we will name **u**.

Since the signature, Σ, and the set of features, **F**, are assumed to be part of the context in which a ψ-clause has meaning, we need to relate Σ and **F** to the domain before a meaning can be assigned to a ψ-clause.

Definition. Given a signature Σ and set of features, **F**, an *interpretation* **I** is a pair $<h, r>$ where

h is an order homomorphism from Σ to $2^{\mathbf{U}}$. That is, h is a function respecting the order of Σ:

(i) $a \leq b$ implies $h(a) \subseteq h(b)$,

(ii) $h(\top) = \mathbf{U}$,

(iii) $h(\bot) = \{\mathbf{u}\}$, and

(iv) $h(a \wedge b) = h(a) \cap h(b)$

r is a function mapping **F** (features) to **R**, the binary relations over the domain. □

We write r[f] to denote the relation corresponding to feature f. Note that $\mathbf{u} \in h(a)$ for all $a \in \Sigma$ follows from this definition.

Our intent is for ψ-terms (and their subterms, as well) to denote sets of objects in U and for ψ-clauses to make statements about domain objects and the relations between them. The domain element **u** is placed in every set h(a) to ensure that each set is non-empty and, thus, that the statement a clause makes is never vacuously true. We will assume that **u** participates in every relation, i.e., that

[3] Deliyanni and Kowlaski discuss the adequacy of binary relations instead of full n-ary relations in the context of a knowledge representation scheme vaguely similar to the one presented here [Deliyanni and Kowalski 1979].

$$<d, u> \in r[f]$$

and

$$<u, d> \in r[f]$$

hold for every domain element d and feature f. The element u might be called a *fantasy object* since it satisfies every relation. The ability to name a specific element satisfying the requirements of a clause will come in handy later but, for the most part, such as in the examples, u will be ignored.

Before we can discuss the meaning of clauses, we need to describe the association between the terms in a clause and domain elements. Since h and r impose structure on the domain, we only want a term t to be a description of an object $d \in U$ if the structure of t — its head symbol and attributes — respects the structure of the interpretation. The first definition describes the association. The second definition describes the requirements of the association for it to even be meaningful. The third definition tells when the association respects the interpretation.

Definition. A *grounding function* g is a partial function mapping addresses to domain elements:

$$g: \mathbf{F}^* \rightarrow \mathbf{U}$$

□

Definition. Let $<\mathbf{S}, \mathbf{A}, \psi, \tau>$ be the set of shared ψ-terms $t_1, \ldots, t_k$. Given an interpretation $I = <h, r>$, a grounding function is said to be *sensible* if the following conditions hold:

(i) g(a) is defined for all $a \in \mathbf{A}$.

(ii) $g(a) \in h(\psi(a))$ when g(a) is defined.

(iii) $g(a_i) = g(a_j)$ whenever $a_i \equiv a_j$.

□

Definition. An interpretation $I = <h, r>$ *satisfies* a grounding function g if

$$<g(a), g(a.f)> \in r[l]$$

whenever g(a) and g(a.f) are defined, where $a \in \mathbf{F}^*$ and $f \in \mathbf{F}$. □

Definition. Let $<\mathbf{S}, \mathbf{A}, \psi, \tau>$ be the set of shared ψ-terms $t_1, \ldots, t_k$ and I be an interpretation. We say that a grounding function g *satisfies* $t_1, \ldots, t_k$ if g is sensible for $t_1, \ldots, t_k$ and I satisfies g. An interpretation **I** *satisfies* $t_1, \ldots, t_k$ if I satisfies every sensible grounding function for $t_1, \ldots, t_k$. □

A grounding function may be thought of as associating a domain object d with every vertex in the graphical representation of the ψ-terms. The object (i) must be in the set of domain objects whose name (from Σ) labels the vertex and (ii) all coreferring subterms must be grounded to the same object for the grounding function to be sensible. For such a grounding function to *satisfy* the ψ-terms, the object must be related to other objects as indicated by the attribute list. When feature f_i is present and has a value t_i, then the $r[f_i]$ property of d must include object d_i, where $d_i = g(t_i)$. Features serve to constrain the set of domain elements intended by a ψ-term. When a feature f_i is absent, then the $r[f_i]$ property of d must include *all* objects $d_i \in \mathbf{U}$ since the values of missing features "default" to $\top$.

Example 48. Consider the singleton set of shared ψ-terms consisting of the ψ-term

X: (manager⇒X)

and an interpretation $\mathbf{I}_1$ in which manager is mapped to a relation m and where $<d_1, d_1> \in m$. Then the following grounding function g

```
g(1⇒) = d1
g(1⇒manager⇒) = d1
```

is sensible since (i) g is defined for all addresses in the term, (ii) no address is more constrained than $\top$ (and d_1 is trivially in the set h($\top$)) and (iii) g maps both corefering addresses into the same object. Function g also *satisfies* this ψ-term since the term describes any object d_i such that d_i has as a manager d_i itself and d_1 is a manager of d_1 in the interpretation. □

Example 49. Consider the set of shared ψ-terms

```
john (owns ⇒ car, loves ⇒ W:woman),
bill (loves ⇒ W).
```

Let these terms be indexed by 0 and 1, respectively. Let the domain include the following objects

```
johnSmith, johnBrown, billJones, fredBaker, fredWilson,
car54, car60, suzieBrown, maryThomas
```

Let h map the symbols in Σ to sets of objects as follows:

```
john → {johnSmith, johnBrown}
bill → {billJones}
car → {car54, car60}
woman → {suzieBrown, maryThomas}
```

Assume that, in the interpretation, there is a straightforward mapping between features and relation names. We will italicize relation names so, for example, r[loves] = *loves*. This domain includes the following relationships, given in infix notation:

```
johnSmith loves suzieBrown.
johnBrown loves maryThomas.
johnBrown owns car54.
billJones loves maryThomas.
billJones loves suzieBrown.
fredWilson loves suzieBrown.
fredWilson loves maryThomas.
fredBaker loves suzieBrown.
fredBaker loves maryThomas.
```

To visualize this interpretation, see Figure 4.10. Then the function

g(1⇛) = johnBrown
g(1⇛owns⇛) = car54
g(1⇛loves⇛) = maryThomas
g(2⇛) = billJones
g(2⇛loves⇛) = maryThomas

is a grounding function that satisfies this set of shared ψ-terms. The grounding function

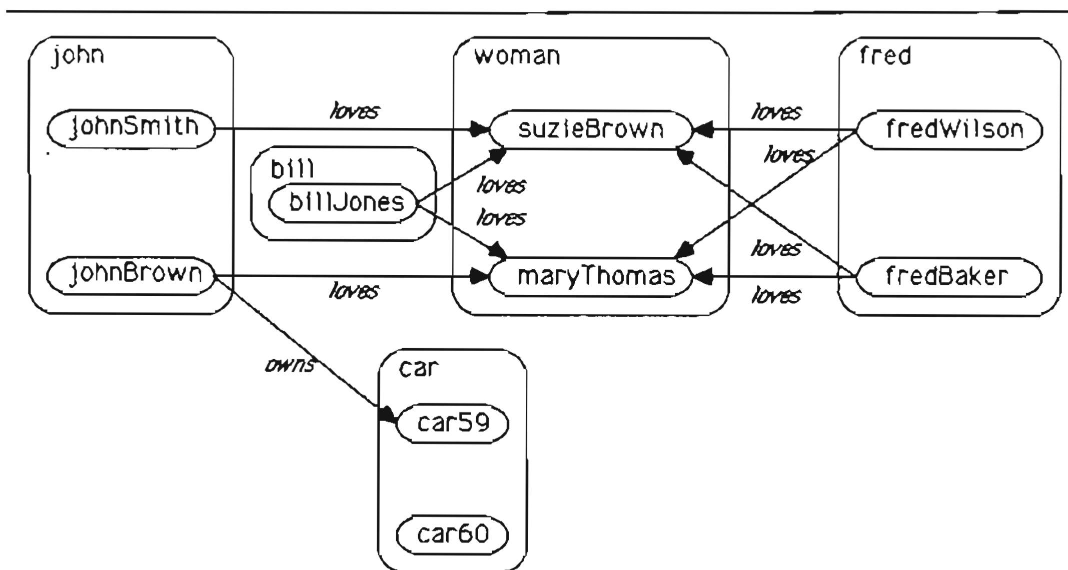

Figure 4.10: An Interpretation

```
g(1⇒) = johnBrown
g(1⇒owns⇒) = car60
g(1⇒loves⇒) = maryThomas
g(2⇒) = billJones
g(2⇒loves⇒) = maryThomas
```

is sensible for this set of shared ψ-terms but does not satisfy it since johnBrown *owns* car54, not car60. In fact, there is no other grounding function that satisfies this ψ-term. □

A grounding function g_1 may be extended to additional addresses, producing a new grounding function g_2 such that $g_2 \supseteq g_1$. Function g_2 is called an *extension* of g_1 and g_1 is called a *restriction* of g_2.

In first-order logic, the notion of *an interpretation satisfying a clause* is introduced in preface to the definition of logical implication. The next definition extends this concept to ψ-logic.

Definition. A clause

$$C: \; t_0 :- t_1, \ldots, t_k.$$

is *satisfied* by an interpretation I = <h, r> if, for any grounding function g_1 satisfying the set of shared ψ-terms $t_1, \ldots, t_k$ and for any sensible grounding function g_2 for the set of shared ψ-terms $t_0, t_1, \ldots, t_k$ that extends g_1 (that is, $g_2 \supseteq g_1$), g_2 necessarily satisfies the set of shared ψ-terms $t_0, t_1, \ldots, t_k$. □

In first-order logic, the quantification of a clause has the general form

$$\forall \vec{X}. \; [\text{body} \Rightarrow (\forall \vec{Y}.\text{head})]$$

where $\vec{X}$ are the variables occurring at least once in the body and the $\vec{Y}$ are the variables that only appear in the head. Because of the way the function of first-order variables is subsumed by the symbols of the signature and because of the more complex

relationship between these symbols and the domain elements, the definition of a ψ-clause quantifies indirectly over the grounding function, using the (approximate) form:

$$\forall g_1. [\text{SatBody}(g_1) \Rightarrow (\forall g_2.\text{SatHead}(g_2))]$$

Example 50. Using the interpretation from Example 49 (see Figure 4.10), consider the clause

```
fred (loves⇒W:woman) :-
     john (loves⇒W), bill (loves⇒W).
```

This clause might be loosely paraphrased as *Fred loves every woman that both John and Bill love*, where *Fred, John*, and *Bill* refer to all individuals whose first names are Fred, John, and Bill, respectively.

Since the symbol `fred` was not used in Example 49, let us extend h as follows:

```
fred → {fredBaker, fredWilson}
```

The clause body

```
john (loves⇒W), bill (loves⇒W).
```

is satisfied by 2 grounding functions. The first (call it g_1) is

```
g(1⇒) = johnBrown
g(1⇒loves⇒) = maryThomas
g(2⇒) = billJones
g(2⇒loves⇒) = maryThomas
```

There are two sensible extensions of g_1 onto the clause head. Call them g_1' and g_1''. The first extension, g_1', is the following grounding function:

```
g(1⇒) = johnBrown
g(1⇒loves⇒) = maryThomas
g(2⇒) = billJones
g(2⇒loves⇒) = maryThomas
g(0⇒) = fredBaker
g(0⇒loves⇒) = maryThomas
```

The function g_1' satisfies the clause since

```
fredBaker loves maryThomas.
```

The second extension, g_1'', is the function

```
g(1⇒) = johnBrown
g(1⇒loves⇒) = maryThomas
g(2⇒) = billJones
g(2⇒loves⇒) = maryThomas
g(0⇒) = fredWilson
g(0⇒loves⇒) = maryThomas
```

The function g_1'' also satisfies the clause since

```
fredWilson loves maryThomas.
```

The second grounding function satisfying the body (call it g_2) is:

```
g(1⇒) = johnSmith
g(1⇒loves⇒) = suzieBrown
g(2⇒) = billJones
g(2⇒loves⇒) = suzieBrown
```

There are also two ways to extend g_2 onto the clause head. Call them g_2' and g_2''. The first, g_2', is:

```
g(1⇒) = johnSmith
g(1⇒loves⇒) = suzieBrown
g(2⇒) = billJones
g(2⇒loves⇒) = suzieBrown
g(0⇒) = fredBaker
g(0⇒loves⇒) = suzieBrown
```

The function g_2' satisfies the clause since

```
fredBaker loves suzieBrown.
```

The second extension, g_2'', is the function

```
g(1⇒) = johnSmith
g(1⇒loves⇒) = suzieBrown
g(2⇒) = billJones
g(2⇒loves⇒) = suzieBrown
g(0⇒) = fredWilson
g(0⇒loves⇒) = suzieBrown
```

The function g_2'' also satisfies the clause since

`fredWilson` *loves* `suzieBrown`.

Consequently, the clause is satisfied by this interpretation.

As this example illustrates, a more accurate paraphrase of the clause might be *Any woman that is loved by a person named John and by a person named Bill is loved by all persons named Fred*[4]. Mary Thomas, who is loved by John Brown and Bill Jones, is loved by both Fred Baker and Fred Wilson. Another woman -- Suzie Brown -- meets the conditions of the body by being loved by both John Smith and Bill Jones. She, too, is loved by both Fred Baker and Fred Wilson. □

The following two definitions extend *satisfaction* to ψ-programs and to ψ-queries.

Definition. Let G be a ψ-program. We say that an interpretation I *satisfies* G when I satisfies all the ψ-clauses in G. □

Definition. An interpretation I *satisfies* a ψ-query Q

$:- \ t_1, t_2, \ldots, t_k.$

if there exists a grounding function satisfying the corresponding shared clause set

$t_1, t_2, \ldots, t_k$

□

[4] Since there is no symbol `person` in the signature, an even more accurate paraphrase might read *Any woman that is loved by any thing named John and...*

4.6.1. Soundness and Completeness

Next, we define *soundness* and *completeness* for proof procedures. These definitions are essentially the same definitions used in first-order logic, adapted to ψ-logic. We begin by defining logical implication ($\models$) and provability ($\vdash$).

Definition. Given a set of ψ-clauses P and a ψ-query Q, we write $P \models Q$ if, for every interpretation **I** satisfying P, **I** also satisfies Q. Likewise, given a set of ψ-clauses P and a ψ-clause C, we write $P \models C$ if, for every interpretation **I** satisfying P, **I** also satisfies C. □

Definition. A *proof rule* is a procedure that takes as input two ψ-clauses (C_1 and C_2) and produces as output a third ψ-clause. If a given rule produces as output the ψ-clause C, we write $C_1, C_2 \vdash C$. A *ψ-theorem prover* (or *ψ-interpreter*) is a program that takes as input a set of ψ-clauses P and a ψ-query Q and produces as output a set of shared ψ-terms. If, given a ψ-query Q, a ψ-interpreter produces as output the set of shared ψ-terms Q′, we write $P[Q] \vdash Q'$. □

Example 51. Let P be the following program

```
p(f=>a).
p(f=>b).
```

Assume all symbols are unrelated, i.e., that Σ is a flat lattice. For the ψ-query Q

```
:- p.
```

our interpreter, described in Chapter 5, produces two outputs, Q'_1:

```
:- p(f=>b).
```

and Q'_2:

```
:- p(f=>b).
```

Thus, we write $P[Q] \vdash Q'_1$ and $P[Q] \vdash Q'_2$. □

Now, we can say what it means for a proof procedure to be sound and/or complete.

Definition. A proof rule is *sound* if $C_1, C_2 \vdash C$ implies $C_1 \cup C_2 \models C$. A ψ-interpreter is *sound* if, for any ψ-query Q, $P[Q] \vdash Q'$ implies $P \models Q'$. □

Example 52. Note that, for program P, query Q, and outputs Q'_1 and Q'_2, from Example 51, our interpreter behaves soundly since

$$P \models Q'_1$$

and

$$P \models Q'_2$$

both hold. □

Prolog interpreters are given a query and a program and, upon finding a proof, print "Yes" and the variable substitution used in the proof. Printing a substitution doesn't work well in ψ-logic, since the coreference and cyclic structure of ψ-terms is difficult for the interpreter to display clearly when several related ψ-terms are displayed in a variable-substitution format. After initially exploring several techniques for displaying answers, we found that it was clearest to display the original query with the results of the various unifications applied to it. Thus, the answer displayed when a proof is found will be a more specific instance of the original query, i.e., if $P[Q] \vdash Q'$ then $Q' \sqsubseteq Q$.

Definition. A proof procedure is *complete* if, for all ψ-queries Q, whenever $P \models Q''$ for some $Q'' \sqsubseteq Q$, we have $P[Q] \vdash Q'$ for some Q' such that $Q'' \sqsubseteq Q'$. □

If some instance Q'' of the query is logically implied by the program, then the interpreter had better produce an output that is at least as general as Q''.

Example 53. Consider program P from Example 51 and let Q''_1 be the ψ-query

```
:- p(f⇒a, g⇒c).
```

and Q''_2 be

```
:- p(f⇒b, g⇒d).
```

Notice that

$$P \models Q''_1 \text{ and } Q''_1 \sqsubseteq Q$$

and

$$P \models Q''_2 \text{ and } Q''_2 \sqsubseteq Q$$

Although our ψ-interpreter is not in general complete, it behaves completely for query Q by producing two outputs Q'_1 and Q'_2. That is,

$$P[Q] \vdash Q'_1 \text{ and } Q''_1 \sqsubseteq Q'_1 \sqsubseteq Q$$

and

$$P[Q] \vdash Q'_2 \text{ and } Q''_2 \sqsubseteq Q'_2 \sqsubseteq Q$$

□

As in logic programming, soundness is required and completeness is desired of a candidate execution environment for Inheritance Grammars.

4.6.2. Soundness of ψ-Resolution

We conclude this chapter by showing that the ψ-clause resolution rule as defined above is sound. We first make several observations and present four useful lemmata as a prelude to the soundness result.

Lemma 2. Let T = <S, A, ψ, τ> and T' = <S', A', ψ', τ'> be two sets of shared ψ-terms such that $T' \sqsubseteq T$. If grounding function g is sensible for T', then g is sensible for T. □

Proof. Let g be any arbitrary grounding function sensible for T'. Since g(a') is defined for all $a' \in A'$ and $A \subseteq A'$, g(a) is defined for all $a \in A$. Since $g(a) \in h(\psi'(a))$ for all $a \in A$ and since $\psi'(a) \leq \psi(a)$ and since h is an order homomorphism, $g(a) \in h(\psi(a))$. Since $g(a_i) = g(a_j)$ for all $a_i, a_j \in A$ whenever $a_i \equiv' a_j$ and since $\equiv \subseteq \equiv'$ (i.e., $\equiv$ is less constraining than $\equiv'$), $g(a_i) = g(a_j)$ whenever $a_i \equiv a_j$. Thus, g is sensible for T.

Example 54. Let's modify the interpretation from Example 49 (Figure 4.10) by adding anouther signature symbol `johnny`. Johnny is to be a nickname for John Brown but not for John Smith so we will add

```
johnny → {johnBrown}
```

to h. The following grounding function g

```
g(1⇒) = johnBrown
g(1⇒owns⇒) = car54
g(1⇒loves⇒) = suzieBrown
```

is sensible for the term:

```
johnny (owns ⇒ car, loves ⇒ woman)
```

The function g is also a sensible grounding function for a more general term

```
john (loves ⇒ woman)
```

Note that g fails to satisfy either of these terms since John Brown does not love Suzie Brown in the interpretation. □

Lemma 3. Let T and T' be two sets of shared ψ-terms.

$$T = t_1, \ldots, t_k$$
$$T' = t'_1, \ldots, t'_k$$

Let T' be more specific than T, i.e., $T' \sqsubseteq T$. Every grounding function g that satisfies T' necessarily satisfies T. □

Proof. If g satisfies T', then g must be sensible for T'. By Lemma 2, if g is sensible for T', then g must be sensible for T. If g satisfies T', then the interpretation must, by definition, satisfy g. Finally, if g is sensible for T and is satisfied by the interpretation, g must satisfy T. □

Example 55. Let

```
T' = johnny (owns ⇒ car, loves ⇒ W:woman),
       bill (loves ⇒ W)
```

and

```
T = john (loves ⇒ woman),
       bill (loves ⇒ woman)
```

Clearly $T' \sqsubseteq T$. The function g

```
g(1⇒) = johnBrown
g(1⇒owns⇒) = car54
g(1⇒loves⇒) = maryThomas
g(2⇒) = billJones
g(2⇒loves⇒) = maryThomas
```

satisfies T'. The grounding function g is sensible for T' and, by Lemma 2, for T. Since g satisfies T', g is satisfied by the interpretation. Thus, g satisfies T. □

Lemma 4. Let $T = \langle \mathbf{S}, \mathbf{A}, \psi, \tau \rangle$ and $T' = \langle \mathbf{S}', \mathbf{A}', \psi', \tau' \rangle$ be two sets of shared ψ-terms such that $T' \sqsubseteq T$. If interpretation **I** satisfies T then **I** satisfies T'. □

Proof. Let g be any sensible grounding function for T'. Since $T' \sqsubseteq T$, g is sensible for T by Lemma 2. Since **I** satisfies T, **I** satisfies g. Thus, since g is sensible for T' and since **I** satisfies g and g was chosen arbitrarily, **I** satisfies T' by the definition of ψ-

clause satisfaction. □

Example 56. Consider the following two (sets of shared) ψ-terms

T = `john (loves ⇒ woman)`

and

T′ = `johnny (loves ⇒ woman)`

Clearly $T' \sqsubseteq T$. First, consider the following grounding function g:

g(`1⇒`) = `johnBrown`
g(`1⇒loves⇒`) = `maryThomas`

This grounding function satisfies T (under the interpretation given earlier) and, as Lemma 4 would seem to imply[5], g also satisfies T′. Now consider the set of shared ψ-terms:

T″ = `johnny (loves ⇒ woman, owns ⇒ car)`

and note that $T'' \sqsubseteq T$. The function g is not even sensible for T″ and so g does not satisfy T″. Since g satisfies T, shouldn't g also satisfy T″ by Lemma 4? Not necessarily; Lemma 4 makes a statment about interpretations satisfying sets of shared ψ-terms. For an interpretation to satisfy T″, every grounding function *that is sensible* for T″ must be satisfied by the interpretation.

So let's modify g so that it is sensible for T″.

g′(`1⇒`) = `johnBrown`
g′(`1⇒loves⇒`) = `maryThomas`
g′(`1⇒owns⇒`) = `car54`

Now g′ is sensible for T″ and satisfies T″ since

[5] Specious reasoning; keep reading.

```
johnBrown owns car54.
```

But what if we had extended g as follows instead?

```
g''(1⇒) = johnBrown
g''(1⇒loves⇒) = maryThomas
g''(1⇒owns⇒) = car60
```

In the interpretation, johnBrown does not own car60, so g'' does not satisfy T''. Since g'' is sensible for T'', Lemma 4 seems to say that it should do so. What's going on here?

The problem now is that g'' does not satisfy T either. The fact that g'' is sensible for T but is not satisfied by the interpretation means that T is not satisfied by the interpretation in the first place. So Lemma 4 doesn't apply. Intuitively, T says that Johnny loves all women and, by ommision of an owns⇒ attribute, that Johnny owns *everything*! In the interpretation we are using, Johnny only loves one woman and owns one car. □

This property of ψ-logic – where missing attributes imply a relationship to all elements in the domain – is both a strength and weakness. The implied value of $\top$ for missing attributes makes it possible to elide unimportant attributes, thereby simplifying ψ-terms while still allowing ψ-clause resolution to proceed. But, on several occasions in the development of the Inheritance Grammar of Appendix 4, missing attributes caused subtle bugs.

The following rule-of-thumb seems to make writing ψ-logic programs easier. *It is okay to omit attributes from terms in a clause body but never omit attributes from a clause head.* This rule only makes sense when there is a specific set of attributes associated with a given head symbol. Only then does it make sense for "an attribute to be omitted accidentally." Of course an association of specific attributes with specific symbols of the

signature is totally absent from the definition of ψ-terms. The association is only in the mind of the grammar writer. In *type checking* a traditional programming language, a symbol can be used correctly or incorrectly. An interesting question is how to modify ψ-logic to capture what appears to be some form of high-level type checking currently being performed by the grammar writer.

The next lemma discusses whether a clause is satisfied when it is subsumed by another satisfied clause.

Lemma 5. Let $C' = \langle \mathbf{S}', s', \mathbf{A}', \psi', \tau' \rangle$ be a ψ-clause:

$$C': \; t'_0 :- t'_1, \ldots, t'_k.$$

and let $C = \langle \mathbf{S}, s, \mathbf{A}, \psi, \tau \rangle$ be a ψ-clause

$$C: \; t_0 :- t_1, \ldots, t_k.$$

such that C' is subsumed by C, i.e., $C' \sqsubseteq C$. If an interpretation **I** satisfies C then **I** also satisfies C'. □

Proof. Let g_1 be an arbitrary grounding function sensible for $t'_1, \ldots, t'_k$ that is satisfied by interpretation **I**. To show **I** satisfies C', we must show that **I** also satisfies any sensible extension of g_1 onto $t'_0, t'_1, \ldots, t'_k$. Let g_2 be an arbitrary extension of g_1 ($g_2 \supseteq g_1$) onto $t'_0, t'_1, \ldots, t'_k$. By Lemma 2, g_1 is sensible for $t_1, \ldots, t_k$ and g_2 is sensible for $t_0, t_1, \ldots, t_k$. Since g_1 is sensible for $t_1, \ldots, t_k$ and is satisfied by **I**, g_1 satisfies $t_1, \ldots, t_k$. By assumption, **I** satisfies C so the definition of clause satisfaction allows us to conclude that **I** must satisfy g_2. Thus, **I** satisfies $t'_0 :- t'_1, \ldots, t'_k$, again by the definition of clause satisfaction. □

Theorem 7. Resolution of ψ-clauses is sound. That is, if an interpretation **I** satisfies clause C_1 and C_2 and they are resolved to produce clause C, then **I** satisfies C.

□

Proof. (original) Assume that clause C_1 = $<S_1, 0, A_1, \psi_1, \tau_1>$ denoted

$$C_1: \; t_0 :- t_1, \ldots , t_i, \ldots , t_j.$$

and clause $C_2 = <S_2, j+1, A_2, \psi_2, \tau_2>$ denoted

$$C_2: \; t_{j+1} :- t_{j+2}, \ldots , t_k.$$

resolve on index i (i.e., $t_i \sqcap t_{j+1} \neq \perp$) to produce the clause $C = <S, 0, A, \psi, \tau>$

$$C: \; t'_0 :- t'_1, \ldots , t'_{i-1}, t'_{i+1}, \ldots , t'_j, t'_{j+2}, \ldots , t'_k.$$

Assume that interpretation I satisfies C_1 and C_2; we must show that I satisfies C.

To show that I satisfies C, we must show that, for every grounding function that satisfies the body of C

$$t'_1, \ldots , t'_{i-1}, t'_{i+1}, \ldots , t'_j, t'_{j+2}, \ldots , t'_k.$$

and for every extension onto the head of C, t'_0, the interpretation satisfies the extension. Let g_1 be an arbitrary grounding function satisfying the body of C

$$t'_1, \ldots , t'_{i-1}, t'_{i+1}, \ldots , t'_j, t'_{j+2}, \ldots , t'_k.$$

and let g_2 be any extension onto t'_0, such that $g_2 \supseteq g_1$. It is sufficient to prove that I satisfies g_2.

Define the clause $C'_1 = <S'_1, 0, A'_1, \psi'_1, \tau'_1>$

$$C'_1 = (\; C_2 \; \{i \leftarrow j+1, \bullet \leftarrow j+2, \ldots , \bullet \leftarrow k\} \;) \sqcap C_1$$

and the clause $C'_2 = <S'_2, i, A'_2, \psi'_2, \tau'_2>$

$$C'_2 = (\; C_1 \; \{\bullet \leftarrow 0, \bullet \leftarrow 1, \ldots , \bullet \leftarrow i\text{-}1, i \leftarrow i, \bullet \leftarrow i+1, \ldots , \bullet \leftarrow j\} \;) \sqcap C_2 (\; \{i \leftarrow j+1\} \;)$$

More intuitively, we may write C'_1 as

$$t'_0 :- t'_1, \ldots, t'_i, \ldots, t'_j.$$

and C'_2 as

$$t'_i :- t'_{j+2}, \ldots, t'_k.$$

Clearly $C'_1 \sqsubseteq C_1$ and $C'_2 \sqsubseteq C_2$. Consequently, by Lemma 5, C'_1 and C'_2 are satisfied by **I**.

The general approach of the proof is as follows: To show that g_2 is satisfied, we first define an extension (called g'_1) of g_1 onto t'_i. Since **I** satisfies clause C'_2, we can conclude g'_1 is satisfied. Then we define an extension (called g''_1) of g'_1 onto t'_0. Since I satisfies clause C'_1, we can conclude g''_1 is satisfied. Finally, by the way we will define g''_1, it is an extension of g_2. Since g''_1 is satisfied, any restriction of g''_1 is also satisfied.

To simplify matters, let us first define a clause $C_R = \langle \mathbf{S}_R, 0, \mathbf{A}_R, \psi_R, \tau_R \rangle$ as:

$$C_R = (\ C_2\,\{i \leftarrow j+1,\ j+2 \leftarrow j+2,\ \ldots,\ k \leftarrow k\}\ ) \sqcap C_1$$

Intuitively, we write C_R as

$$C_R:\ t'_0 :- t'_1, \ldots, t'_i, \ldots, t'_j, t'_{j+2}, \ldots, t'_k.$$

Clause C_R is so named because it contains all *relevant* addresses. Clearly, $\mathbf{A}'_1 \subseteq \mathbf{A}_R$, $\mathbf{A}'_2 \subseteq \mathbf{A}_R$, and $\mathbf{A} \subseteq \mathbf{A}_R$. Also note that $\equiv_R$ contains all the coreference information from C'_1, C'_2, and C, i.e.,

$a \equiv'_1 b$ if and only if $a \equiv_R b$ for $a, b \in \mathbf{A}'_1$,
$a \equiv'_2 b$ if and only if $a \equiv_R b$ for $a, b \in \mathbf{A}'_2$, and
$a \equiv b$ if and only if $a \equiv_R b$ for $a, b \in \mathbf{A}$.

Finally note that

$\psi'_1(a) = \psi_R(a)$ for all $a \in \mathbf{A}'_1$,
$\psi'_2(a) = \psi_R(a)$ for all $a \in \mathbf{A}'_2$, and
$\psi(a) = \psi_R(a)$ for all $a \in \mathbf{A}$.

Let $x \Rightarrow a$ denote an address that begins with index x. We define the grounding function g'_1 as follows:

$$
\begin{array}{ll}
g'_1(a) = g_1(a) & \text{when } g_1(a) \text{ is defined, otherwise} \\
g'_1(i \Rightarrow a) = g_1(b) & \text{when } i \Rightarrow a \equiv_R b \text{ and } g_1(b) \text{ is defined, otherwise} \\
g'_1(i \Rightarrow a) = g_2(b) & \text{when } i \Rightarrow a \equiv_R b \text{ and } g_2(b) \text{ is defined, otherwise} \\
g'_1(i \Rightarrow a) = u & \text{when } i \Rightarrow a \text{ is in } \mathbf{A}_R \text{ but is not defined above, otherwise} \\
g'_1(a) = \textit{undefined}. &
\end{array}
$$

Next, define the grounding function g''_1 as follows:

$$
\begin{array}{ll}
g''_1(a) = g'_1(a) & \text{when } g'_1(a) \text{ is defined, otherwise} \\
g''_1(a) = g_2(a) & \text{when } g_2(a) \text{ is defined, otherwise} \\
g''_1(a) = \textit{undefined}. &
\end{array}
$$

Clearly g''_1 is an extension of g'_1 and g'_1 is an extension of g_1, i.e., $g_1 \subseteq g'_1 \subseteq g''_1$. Also, these definitions are constructed in such a way that the restriction of g''_1 onto the domain of g_2 is g_2 itself, i.e., $g''_1(a) = g_2(a)$ when $g_2(a)$ is defined, since $g_1 \subseteq g_2$.

To show that g'_1 is satisfied, we must first show that g'_1 is sensible for clause C'_2. Intuitively, g'_1 is defined over $\mathbf{A}'_2$ because g_1 is defined over addresses beginning with indices j+2 though k and the definition of g'_1 adds definitions for all addresses beginning with index i. The ordering h is also respected, i.e., $g'_1(a) \in h(\psi'_2(a))$, since the value of $g'_1(a)$ is taken from either $g_1(a)$ or $g_2(a)$, which respect h, i.e., $g_1(a) \in h(\psi(a))$ and $g_2(a) \in h(\psi(a))$ and $\psi(a) = \psi'_2(a)$ when both are defined, or the value of $g'_1(a)$ is $\mathbf{u}$, which trivially respects h since $\mathbf{u} \in h(\sigma)$ for all $\sigma \in \Sigma$. To see that the coreference relation is respected, i.e., $g'_1(a) = g'_1(b)$ whenever $a \equiv'_2 b$, note that the definition of g'_1 respects $\equiv_R$ and that $\equiv'_2$ and $\equiv_R$ agree when both are defined. So, g'_1 is an extension of g_1 sensible for clause C'_2. Since C'_2 is satisfied, we conclude that g'_1 is satisfied.

To show that g''_1 is satisfied, we must first show that g''_1 is sensible for C'_1. Intuitively, g''_1 is defined on $\mathbf{A}'_1$ since g'_1 is defined over addresses beginning with indices 1, ...

, i, ... , j, j+2, ... , k and the definition of g''_1 extends g'_1 by adding the remaining mappings in g_2, which include the addresses beginning with index 0. The order is respected, i.e., $g''_1(a) \in h(\psi'_1(a))$, since the value of $g''_1(a)$ is taken from either $g'_1(a)$ or $g_2(a)$. Since $g'_1(a) \in h(\psi_R(a))$ and $g_2(a) \in h(\psi_R(a))$ and $\psi'_1(a) = \psi_R(a)$ when $\psi'_1(a)$ is defined, $g''_1(a) \in h(\psi'_1(a))$. The coreference relation is respected, i.e., $g''_1(a) = g''_1(b)$ whenever a $\equiv'_1$ b, for the same reason g'_1 respected $\equiv'_2$. Namely, g''_1 takes its values from g_2 and, indirectly, from g_1, and g_1 and g_2 both respect $\equiv_R$ which agrees with $\equiv'_1$ whenever both are defined. So, g''_1 is an extension of g'_1 sensible for clause C'_1. Since C'_1 is satisfied, we conclude that g''_1 is satisfied.

Recapping, g'_1 is an extension of g_1 sensible for clause C'_2 and so g'_1 is satisfied. Likewise, g''_1 is an extension of g'_1 onto clause C'_1, so g''_1 is also satisfied. Thus, since g_2 is a restriction of a satisfied grounding function (g''_1), we conclude g_2 is satisfied. Since we chose g_1 as an arbitrary grounding function satisfying the body of C and g_2 as an arbitrary extension of g_1 sensible for the head of C, we have shown that C is satisfied by interpretation I. Thus, the resolution rule is sound. □

We have implemented a ψ-interpreter (in Smalltalk-80) which we will describe in the Chapter 5. We conjecture that our implementation is sound but the system is far too large for any formal proof. Our interpreter in certainly not complete; we will however take up issues of complete strategies in Chapter 6.

Part 3
Implementation

Chapter 5:
Implementations of Inheritance Grammar

5.1. Introduction

To test our ideas about Inheritance Grammars and their implementation and to experiment with sample Inheritance Grammars we have produced two resolution-based theorem provers to execute IGs. Both systems are written in Smalltalk for the Tektronix 4317 workstation and both use a depth-first evaluation strategy, which makes them incomplete by the definition of *completeness* given in Chapter 4.

The first implementation of Inheritance Grammar is an interpreter that does not handle ψ-term predications. That is, head and body literals look like first-order literals with ψ-terms as arguments. This system was written first and proved useful in evaluating the formalisms in spite of this limitation. The interpreter includes a window-oriented grammar development environment, several useful built-in predicates, and list and DCG-like notational abbreviations. To speed execution, clauses are parsed and translated into an internal representation before being used. This representation closely mirrors the textual representation of the clauses. To facilitate rapid grammar prototyping, the parsing and translation is done incrementally.

The second system implements full Inheritance Grammar, with ψ-term predications. This system is based on the stack architecture used to implement Prolog efficiently. Each clause is viewed as a procedure and is compiled into a sequence of operators for an abstract machine. When executed, these operators allocate *activation records*, copy terms, and perform unifications. See [Maier and Warren 1988] for an

exposition of Prolog execution algorithms.

To differentiate between the two implementations, we call the first the *interpreter* system and the second the *compiler* system. While the compiler system is faster than the interpreter, it has not been integrated with the window-oriented user interface.

The purpose of this chapter is to describe both of these implementations and the user interface for the interpreter system. In order to provide an overview of the techniques employed in the systems, the discussion is at a fairly high level of abstraction. To describe the systems in detail would require a detailed digression into the particulars of programming in Smalltalk.

5.2. The Interpreter System

The interpreter implements the full LOGIN language (as documented in [Ait-Kaci and Nasr 1986]) as well as several useful extensions. A LOGIN program is similar to a ψ-program in that it consists of clauses and ψ-terms are used in place of first-order terms. The primary difference is that LOGIN does not allow ψ-term predications. Instead, each clause is composed is composed of literals, which have the form $p(X_1, \ldots, X_n)$ where p is the predicate name and the X_i are ψ-terms.

For example

```
p(a(m⇒s, n⇒t)) :- q(X:b(m⇒s), c), r, s(X, d(n⇒t), X).
```

is a legal LOGIN clause, as well as a legal ψ-clause. However, the following clause

```
a(m⇒s, n⇒t) :- X:b(m⇒s), c, X, d(n⇒t), X.
```

is a legal ψ-clause but is not a legal LOGIN clause because it uses ψ-terms at the predicate level.

The parser is fairly robust, displaying descriptive syntax error messages. List notation – where a list of ψ-terms is enclosed in brackets – and grammar rule notation – where the punctuation symbol `-->` is used in place of the punctuation `:-` in a clause – are supported by the parser. (Examples of list and rule notation are given later in this chapter when the the syntax of IGs is discussed in detail.) The lexical analyzer supports comments, handles numerals, and supports arbitrary length strings containing non-alphanumeric characters.

The interpreter implements several built-in predicates. The arithmetic predicates support numeric comparison and simple computation on integers. The output predicates can be used by the grammar to produce responses to natural language input. The predicates `assert`, `call`, and `not` can be used to modify and access a database containing semantic information. For example, an Inheritance Grammar might process an English declarative statement by translating it into a clause and `assert`-ing that clause into the database. A question would be translated into a goal which would then be `call`-ed against that database. Finally, the `write` predicate would be used to print a response to the question.

The grammar development environment is window-oriented and menu-driven in the spirit of most Smalltalk applications. Grammar files are stored on the disk in a machine-readable, semi-human-readable format. One menu entree allows the user to *file in* a grammar. Another entree allows him to open a *browser* on the grammar. When a grammar is read in, it is parsed and translated into an internal data representation. To facilitate incremental translation, the user may break the grammar into a number of *pages*. The user may move around from page to page using the browser and may open several browsers on the same grammar for viewing different parts of a single grammar simultaneously. To change the grammar, the user makes the changes to the

textual representation using the standard Smalltalk editor. (He can cut and paste blocks of text and move text from one page to another.) When the changes are complete, only the modified text is reparsed and retranslated.

Another menu entree allows a *dialog* window to be opened. The user can enter natural language input into this window. This input is then broken into lexical tokens (using the same lexical analyzer) and the grammar is called to analyze the string. Presumably, the rules of the grammar will produce some response, which is then inserted into the dialog window.

The signature created by the ordering statements in the grammar can be viewed by using another menu entree. The partial order is displayed graphically and menu entrees allow LUBs and GLBs to be computed, dynamically adding new elements to the display as required. The user can also modify the symbols and their ordering, if desired, through menu options.

The interpreter produces descriptive error messages during parsing, execution, and use of the environment. For the lexical analyzer and parser, the messages include the location of the error.

For a symbol table, the interpreter uses a package, called `Lattice.st`, which implements an *abstract data type* (in the sense of [Goguen, et al. 1978]) for manipulation and display of the signature ordering. This code provides classes called `Lattice` and `LatticeItem`. Each instance of `Lattice` is actually a partial order. Each instance of `LatticeItem` is an element of such an order. Messages can be sent to a lattice object to add new symbols and to add new relationships between elements. The ordering statements in the grammar use these messages to build a partial order. Other lattice messages allow GLBs and LUBs to be found and returned. When the GLB of a set

of items is requested from a lattice object that does not contain a unique GLB, it is created and added to the order. This addition is done incrementally and dynamically as GLBs and LUBs are requested. The lattice protocol also supports *maybe sets*, which are defined in the discussion of the compiler. The window-interface provided for lattice objects will be discussed below, when we discuss the user interface for the interpreter.

The interpreter represents each ψ-term as an object with 4 instance variables, i.e., a record with 4 fields. The first is the root symbol of the term and is implemented by a pointer to a `LatticeItem` object. The second is an `IdentityDictionary` mapping features (i.e., `Strings`) into other ψ-term objects. The third is a `String` giving the variable name associated with the term, if any. This string is used only during the printing of ψ-terms so that the user sees the variables he used, rather than machine-generated names. The fourth field is a *coreference* pointer. Initially, this pointer is nil; coreference of ψ-terms in an input program is captured by shared subterms.

The core of the interpreter is the ψ-term unification algorithm. It is a destructive operation: it is passed two ψ-term objects and returns a pointer to the result object (if the unification succeeds). This result object is constructed from the arguments by altering their fields. When two terms are unified, the coreference field of one is used as a forwarding pointer to the second, which is modified to become the result of the unification. This modification involves finding the GLB of the root symbols in the signature lattice and recursively unifying any subterms. The results of these recursive unifications are then moved into the result term. By always following any non-nil coreference pointers, a reference to either of the original two arguments to the unification will end up at the same term, the result of the unification. See [Ait-Kaci and Nasr 1986] for the complete algorithm.

To take a simple example, consider the unification of two ψ-terms

X: f (m⇒a)

and

Y: g (m⇒b, n⇒c)

The internal representation of these terms is shown in Figure 5.1. (To simplify the figure, pointers to LatticeItems and Strings have been replaced with the corresponding strings themselves.) Assume that neither fg = f∧g nor ab = a∧b is ⊥ in the signature. The result of the unification, shown in Figure 5.2, is

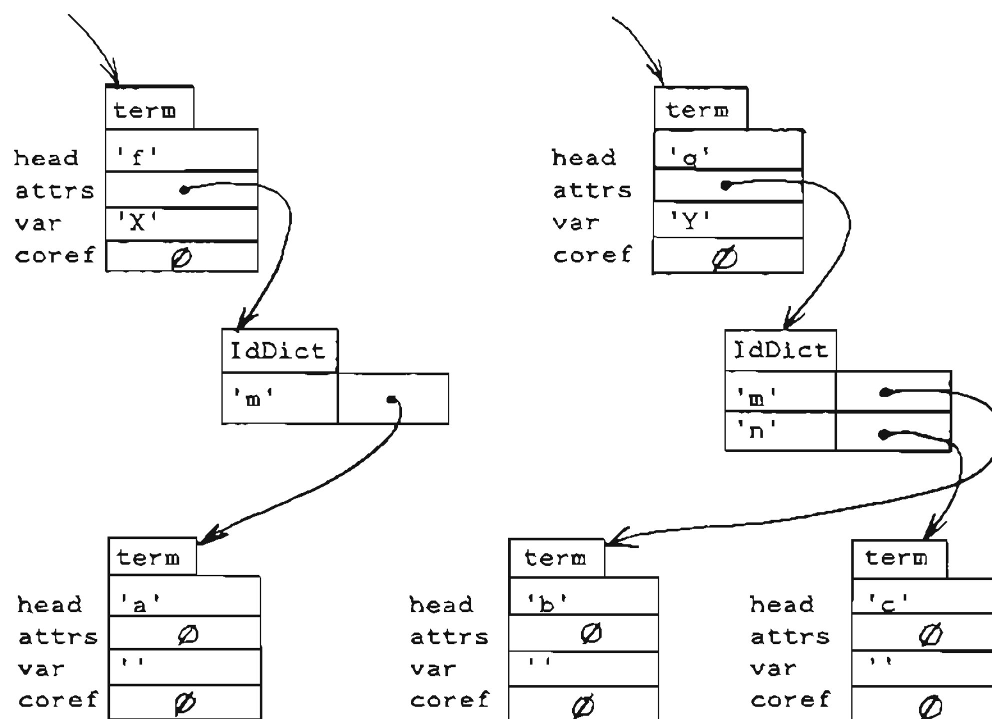

Figure 5.1: The Representation of Two ψ-Terms

```
X:fg(m⇒ab, n⇒c)
```

Since unification overwrites the terms in place, in many cases it can complete without allocating any new objects, as this example illustrates. New objects must be created, however, when new elements are added to the signature or when the `IdentityDictionaries` change substantially.

Given this unification algorithm, the interpreter works as follows. A goal list is satisfied by satisfying each of its literals in turn. A literal is satisfied by first identifying the clauses in the database with a matching predicate name. The clauses of the

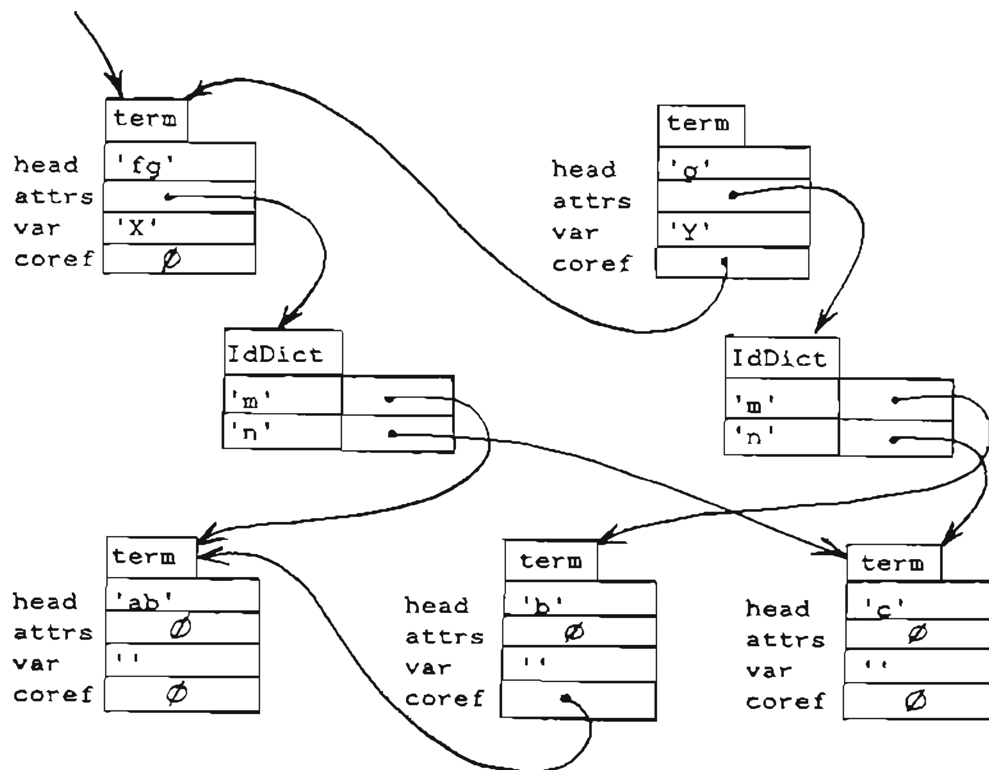

Figure 5.2: The Result of their Unification

database are stored in a dictionary mapping predicates to `OrderedCollections` of clause objects to facilitate this retrieval. Each clause is then copied. Since clauses are, in general, cyclic data structures, this copying can be time consuming and is improved in the compiler. Next, the goal literal is unified with the clause head, which generally modifies many terms in the body. The literals of the body, if any, are prepended onto the list of literals remaining in to be satisfied and the evaluation procedure is called recursively. When all the literals of a goal are satisfied, answers are displayed on the System Transcript, along with execution-timing information.

When no clause can be found to satisfy a goal list, failure occurs, necessitating backtracking. The unification algorithm is constructed so that a *trail* is kept of all terms modified. Using this trail, it is possible to undo the effects of several clause invocations and backup to a previous goal list. A new clause is then selected.

5.3. The Compiler System

The compiler implementation executes full Inheritance Grammar, allowing ψ-term predications. The compiler system is faster than the interpreter, but the window-oriented user interface has not yet been adapted to use the compiler.

The language accepted by the compiler system is very similar to the language accepted by the interpreter system. Beside allowing ψ-term predication, the major differences are in the non-logical extensions. The compiler supports the *cut* operator (!) but the interpreter does not[1]. The interpreter supports `assert`, `call`, and `not`; the compiler does not, although there is no fundamental reason why these operators could

---

[1] *Cut* has proven useful for many users of depth-first execution systems. Unfortunately, the interpreter constructs and modifies goal lists explicitly, making the implementation of *cut* in the interpreter virtually impossible.

not be added to the compiler system. There is also a slight difference in the translation of DCG rules into ψ-clauses.

Both compiler and interpreter use the same lexical analyzer and lattice packages. The parser for the compiler system is a slightly modified version of the parser used in the interpreter system.

A grammar is compiled in one pass by sending it as a string in a message to a class called NewProgram returning a program object. Queries are executed by sending them as strings to this program object (A trivial ψ-program execution is shown in Figure 5.3). Adapting the window-oriented user interface to accommodate compilation would present no fundamental difficulty since the interaction between the user interface and the interpreter is very similar to protocol provided by the compiler system.

The representation of ψ-terms in the compiler system is much more compact than in the interpreter system. In the interpreter, the features of a ψ-term are stored as a set. Unification involves merging the sets of features for the two argument terms which, even if the sets are ordered, is time consuming. In the compiler, ψ-terms are represented as records in which features are stored at known, fixed offsets. In addition to containing a field for each subterm, the record also contains fields for the term's head symbol and coreference pointer. The unification algorithm first looks at the head symbols of the two ψ-term arguments and finds their GLB in the signature. It then quickly runs through the subterm fields, calling itself recursively for each pair of subterms.

But how can fixed offsets be assigned to features? Consider the following grammar fragment:

```
p(l=>..., m=>...) :- ...
... :- ..., q(m=>..., n=>...), ...
```

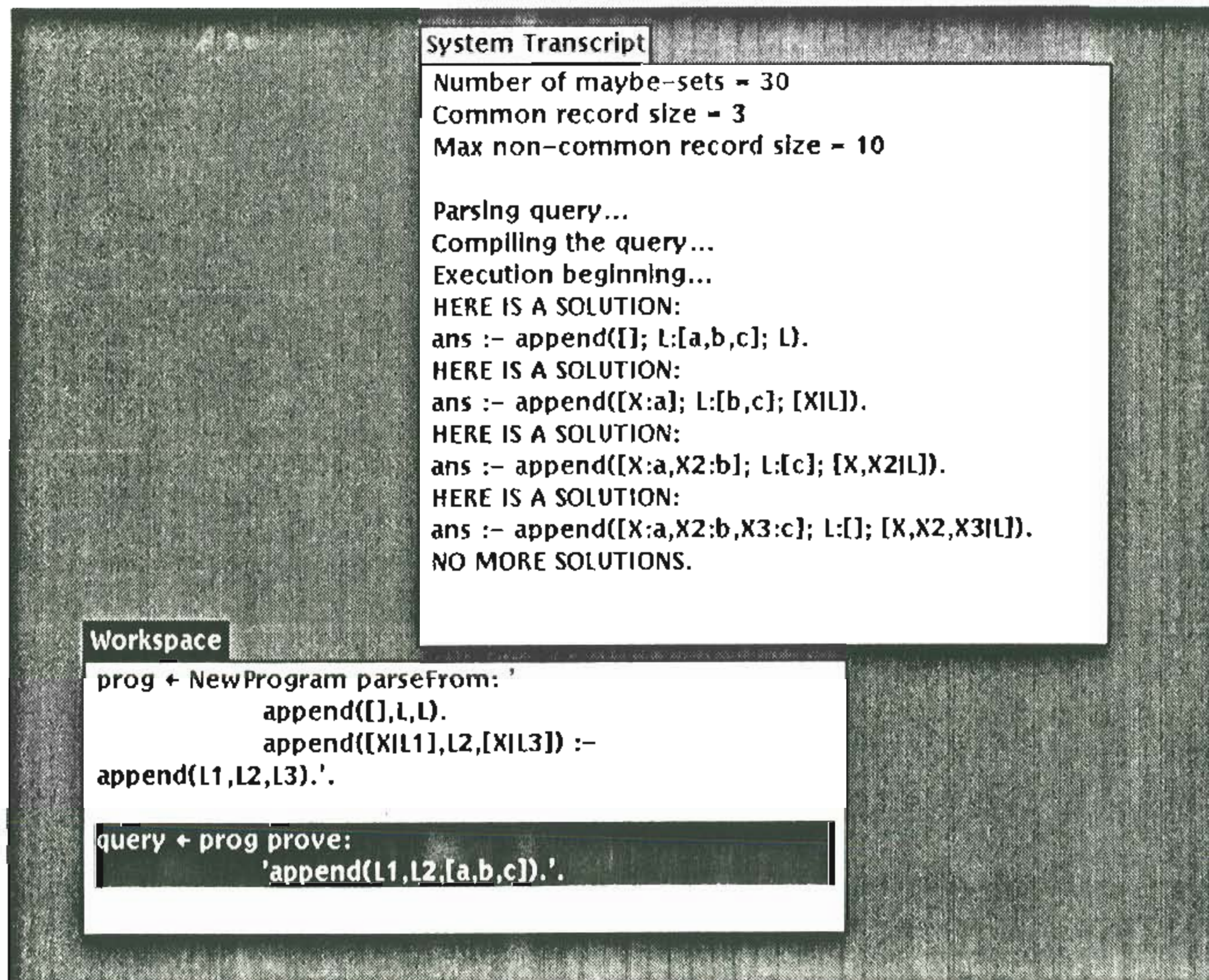

Figure 5.3: Using the Compiler

(The other literals and subterms are unimportant.) Any literal in any clause body (e.g., the q literal) might need to be solved and any rule (e.g., the p clause) might be used, so we might have to unify the q and p literals during execution. The two literals may or may not successfully unify, depending on whether or not p and q are related in the signature.

It would seem that, at compile time, we would need to allocate enough fields in the p and q records to accommodate all possible features. In this case, it would seem that both records would need fields for l, m, and n. Since any body literal might be unified with any head literal, every ψ-term would also need fields for l, m, and n.

Given two ψ-literals at compile time, we can never say whether they will unify successfully at run time, since the goal literal may have been modified through previous unifications at run time. But we can look at two ψ-terms and very often say that they definitely cannot be unified successfully at run time. If, in this example, the symbols p and q are unrelated in the signature, then the unification must always fail. This property of un-unifiability is the basis of the technique we use to compile ψ-terms into fixed-size records. The method is based on the concept of *maybe sets*.

After the grammar has been parsed, the ordering statements are processed to yield the signature. The symbols of the signature are then partitioned into several *maybe sets*. Every symbol (except $\top$ and $\bot$) belongs to exactly one maybe set. The partition is done in such a way that symbols in one maybe set might be comparable or have non-$\bot$ GLBs or non-$\top$ LUBs, but two symbols in different maybe sets are definitely unrelated and always have only $\bot$ as GLB and $\top$ as LUB.

As an example, consider the following ordering statments:

```
r < p.
{r, s} < q.
{t, u} < r.
u < s.
c < a.
c < b.
{d, e} < c.
```

These symbols can be partitioned into 2 maybe sets:

```
{p,q,r,s,t,u}
{a,b,c,d,e}
```

Call the first set number S_1, and the second set S_2. Clearly these sets meet the constraint that, in this ordering, elements of S_1 are completely unrelated with elements in S_2. Note that element p and q have been put in the same maybe set even though they are unrelated and happen to have no LUB.

In constructing the record structure for terms we will build one record structure for all terms whose heads are in maybe set S_1. We will construct a second record format for the other terms, terms whose head symbols are in maybe set S_2. For example, assume the following two ψ-terms appear in the grammar:

```
p (l⇒..., m⇒...)
q (m⇒..., n⇒...)
```

Since the labels l, m, and n are all used, the compiler will allocate a field for each in the record format shared by these two terms. (For simplicity, assume that no other features appear elsewhere in terms with heads from S_1. The compiler will pick an order for these features, such as:

```
l
m
n
```

Thus, the representation of the first term will be a record with 3 fields, the last of which will be empty. The representation of the second term will also be a record with 3 fields;

the first will be empty. If, in the course of executing the grammar, we must unify these two terms, the merging of the features is done in linear time by visiting each field in turn.

What if we encounter a ψ-term with no head, such as the following?

(j⇒..., m⇒...)

Since it might be unified with either of the above two terms at run time, we must add a field for j to the l-m-n record format described above. Since this term might also be unified with terms in other maybe sets, the compiler must ensure that they also have fields j and m and that they be at the same offset within the record.

To handle features like j and m, each record will have a number of *common* fields, as well as *local* fields. In this case, fields j and m must be placed in the common fields and fields l and n will remain local fields. There will be a single block of common fields for the entire grammar, shared by all records, and a block of local fields for each maybe set. The representation of a term such as:

p (l⇒..., m⇒...)

will consist of the common fields, which are always listed first, followed by the local fields:

j
m
-
l
n

The representation of a term such as:

(j⇒..., m⇒...)

will consist of just the common fields (even though it might be unified with a term with

common and local fields):

```
j
m
-
```

These two terms can be unified quickly by comparing the common fields first, and then copying the local fields from the first term to the result.

Given this representation of ψ-terms, how are clauses represented? Each clause is compiled into a sequence with the following general format:

```
Activation Record Size
Head Pointers
Head Patches
Body Pointers
Body Patches
Body
```

As an example, we will consider the clause:

```
X:f(m=>Y:g(n=>X, p=>Y)) :-
            f(m=>X), Y, !, h(q=>X).
```

Assume that f, g, and h are all placed in distinct *maybe sets* and that the feature labels are assigned offsets as follows:

m	in field 1
n	in field 1
p	in field 2
q	in field 1

This clause is represented as indicated by Figure 5.4. We will discuss it in detail in the following paragraphs.

As mentioned above, a stack of activation records is maintained at run time. When a clause is executed, an activation record will be pushed onto the stack. This record contains several fixed fields, along with a variable number of slots (the local variables). The first item in a clause representation is an integer telling how many of these

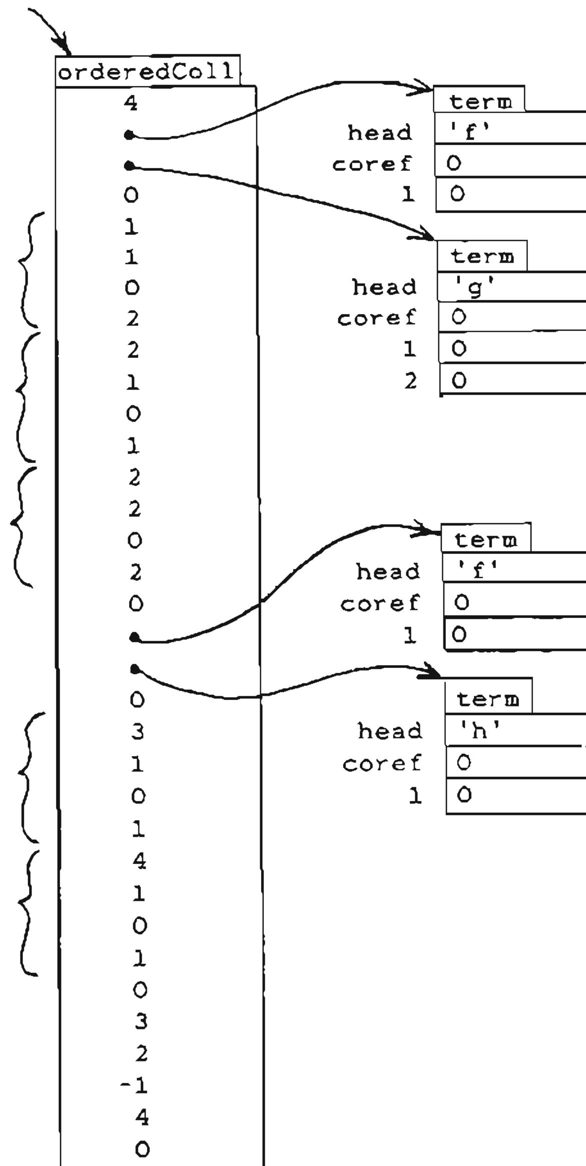

Figure 5.4: The Representation of a ψ-Clause

slots will be needed in the activation record when this clause is executed. In our example clause, the first element indicates that the activation record will need 4 slots.

Since unification of ψ-terms is a destructive procedure that modifies both arguments, the ψ-terms comprising the head and body of a clause must be copied from the database. At run time, we first copy the head ψ-term and then perform the unification with a goal literal. Only if this unification succeeds do we need to copy the body. (The interpreter copies the entire clause before testing the head, which introduces inefficiencies.) The head pointers and head patches are used to do the copying of the head ψ-term. Here is how they work.

The head pointers (there may be several) all point to acyclic terms with no shared subterms. In our example, we have two head pointers, following the activation record size (4). The head of a ψ-clause will in general be cyclic, but experience with the interpreter showed that it is expensive to copy cyclic data structures. At compile time, the term is decomposed into a number of acyclic ψ-terms, which are quickly copied into slots in the activation record. In our example, the two pointers to acyclic terms used in the head would be (deep) copied into the first two slots of the activation record. The head patches are used to modify these terms to add in the appropriate cyclic links.

A head patch (roughly) consists of a sequence of integers. These integers specify (1) a slot in the activation record, (2) a path within the term pointed to by this slot, and (3) another slot in the activation record. Each patch causes one link to be added to a term. The first slot and the path give to location which is modified to point to the second term. There may be zero or more head patches in a clause. In our example, there are 3 head patches. (Head and body patches are indicated with braces in the figure.) The first head patch says to take the record pointed to by slot 1 and modify its first field to point to the record pointed to by slot 2. *After* all the head patches are executed, the cyclic ψ-term representing the head is pointed to by the first activation record slot.

The unification is then performed. If successful, the terms pointed to by the body pointers are (deeply) copied into the remaining activation record slots and the body patches are executed to build the cyclic structure of the ψ-terms in the body, just as the head patches were executed. In our example, there are two body terms and two body patches. Building a representation of a clause that is suitable for the execution of the patches in the manner just described is complicated by the fact that the head and body usually share a number of common subterms. Of course, the subterms that are shared must be constructed when the head is constructed. The difficulty comes in both identifying these subterms and selecting a good strategy for building them. Since copying acyclic terms is faster than executing patches to build equivalent terms (there is an additional layer of interpretation while processing the patches), the compiler tries to minimize the number of patches necessary.

After the body has been constructed, it can be executed. The *body* portion of the clause representation consists of a sequence of integers, one for each literal in the body. In our example, the body consists of

```
3  2  -1  4
```

Each integer is an offset into the activation record, from which a pointer to the appropriate ψ-term can be obtained. A distinguished integer (-1) is used to represent the *cut* operator. Nil values (0) are used in the clause representation to help identify the various parts just described.

Backtracking involves undoing changes that have been made to ψ-terms and trying other clauses. In the compiler, as in the interpreter, a record of the effects of unification is maintained in a *trail*. A trail is a list that records each change made by the unifier along with enough information to undo that change.

The activation records used by the compiler system have the following fields:

```
callingLiteral
callingAR
callingClause
callingClauseIndex
backtrackAR
clauseList
clauseListIndex
beginningTrail
(slots for ψ-term pointers)
```

The fields in the activation record are quite similar to those used in Prolog implementation and are discussed in the next paragraph. (See [Maier and Warren 1988] for a thorough discussion of activation records and Prolog implementation techniques.)

When we have a new literal within a goal that is to be solved, we allocate an activation record. The `callingLiteral` field points to the term representation of that literal. The `callingAR` points to the activation record associated with that goal; on successful completion of the current clause, we will return to that activation record for the next goal literal. The `callingClause` points to the representation of the clause containing the `callingLiteral`; `callingClauseIndex` in the index into that clause of the next goal literal to be solved. Should failure occur, `backtrackAR` points to the activation record of a goal with more clauses to try. The `clauseList` and `clauseListIndex` point to the next clause to try in solving `callingLiteral` in case a subsequent failure causes backtracking to this activation record. The `beginningTrail` is an index into the trail that is used to undo any changes to bring the state back to the point when this activation record was first allocated. The remaining slots are used to build the cyclic structures representing a clause, as described above.

5.4. The Grammar Development Environment

This section describes the window-oriented user interface for the interpreter. The interpreter and interface consist of 3 files:

```
Lattice.st
Login.st
LoginLogo.form
```

The file `Lattice.st` contains 4 classes constituting a subpackage that handles all lattice operations, such as computing LUBs and GLBs. The file `Login.st` contains the classes comprising the interpreter and interface. It needs the the Lattice package to execute properly.

To activate the system, the following message is sent:

```
Program new edit
```

This message causes a new interpreter (object) to be created and the Inheritance Grammar logo to appear. The logo has an associated middle button menu with the following entrees (see Figure 5.5):

```
open rule browser
open dialog browser
open signature window
file in
file out
```

The `file in` selection prompts for the name of a file containing an IG. Appendix 1 lists such a file, `demoGrammar.ig` the name of a file containing a small Inheritance Grammar.

Next, the `open rule browser` selection is used to frame a window for browsing the clauses and ordering statements comprising the grammar. This window is a variation of a `Workspace`, so the left button menu can be used for text selection and the

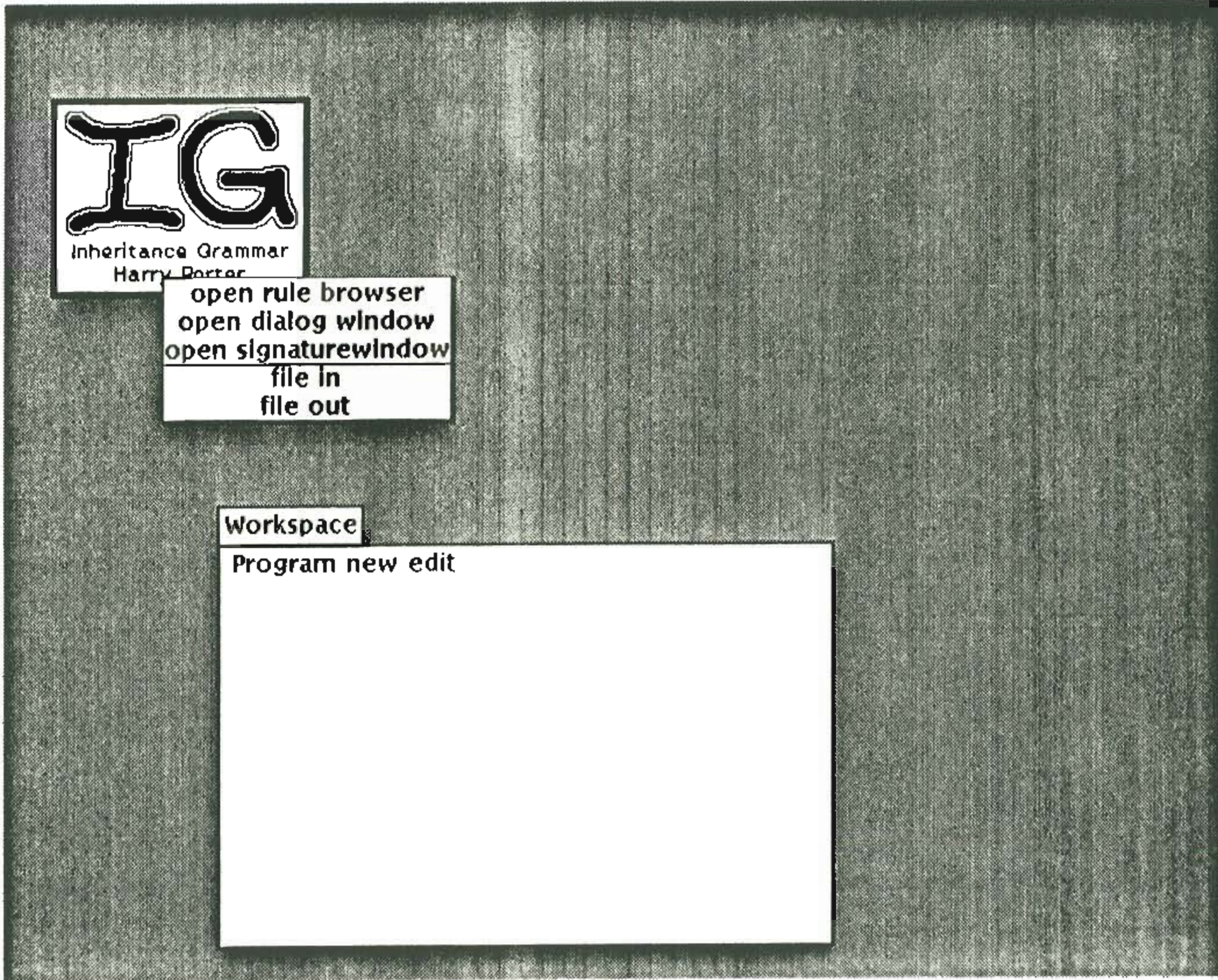

Figure 5.5: Bringing Up The IG System

middle button menu includes the standard entrees for text editing. Figure 5.6 shows a rule browser and its middle button menu, with the following entrees:

```
again
undo
copy
cut
paste
next
previous
first
last
accept
```

Several interpreters may be active simultaneously. When each is initiated, a logo appears on the screen. There is, at any one time, only one Inheritance Grammar associated with each interpreter, although as various commands are executed, this grammar changes. The menu associated with each logo is used to access the current grammar associated with the corresponding interpreter. In addition, there may be several rule browsers active for each grammar, allowing multiple views of a large grammar.

Each grammar is broken up into an ordered sequence of *pages*. Each page contains several clauses and/or ordering statements, but any clause or ordering statement must lie entirely within one page. The grammar writer may distribute clause and ordering statements across pages arbitrarily.

The grammar is divided into pages so that the time needed for the parsing and translation, which must occur every time the grammar is modified, can be reduced. Each rule browser looks at one page of the grammar at a time. When a change is made to the text of the page in a rule browser, only that page is reparsed and retranslated.

The page mechanism also makes it easier to organize large grammars. The example grammar contains six pages, identified in the comments as:

Rule Browser

```
/**********  IS-A hierarchy  **********/

{a,b,c,d,f} < grade.
{maier,kieburtz} < faculty.
{smith,jones} < student.
{student,faculty} < person.
{cs100,cs101} < course.
```

Rule Browser

```
/**********  syntax section **********/

s(Predicate) -->
        np(Subject),vp(Subject,Predicate,singular),
        {addFact(Predicate)}.
s(Predicate) -->
        [did],np(Subject),vp(Subject,Predicate,infinitive),
        {yesNo(Predicate)}.
vp(Subject,Predicate,VForm) -->
        [V],{verb(V,Verb,VForm),
        verbMeaning(Verb,Predicate,[np(Subject)|Complements])},
        vpComplements(Complements).
vpComplements([np(Complement)|Complements]) -->
        np(Complement),vpComplements(Complements).
vpComplements([]) --> .
vpComplements([in(Complement)|Complements]) -->
        [in],np(Complement),vpComplements(Complements).
```

again
undo
copy
cut
paste
next
previous
first
last
accept

Figure 5.6: A Rule Browser

```
Syntax Section
Semantic Processing Section
Generation Grammar
Lexicon
IS-A Hierarchy
Knowledge Base
```

These pages reflect a division of the grammar into (1) the clauses associated with the syntax, (2) the clauses associated with semantic processing, (3) the clauses associated with generating and writing out responses, (4) the clauses comprising a tiny lexicon, (5) the ordering statements, and (6) the clauses comprising a small relational database, respectively.

The middle button menu entrees `next` and `previous` can be used to move back and forth through the pages in the grammar. The entrees `first` and `last` can be used to get to the beginning and the end of the grammar. Changes may be made to a page by modifying the text and then `accept`-ing the page, which causes reparsing and retranslation. (Accepting the first page of this grammar, for example, takes under 2 seconds.)

The `file out` entree on the main interpreter menu can be used to write the grammar out to a file after changes have been made. It will prompt for a file name, defaulting to the last file read in.

Selecting the `open dialog window` entree on the logo menu allows a dialog window to be framed on the screen. This window is a `Workspace` window and input to be parsed by the Inheritance Grammar can be typed and edited using the standard workspace text manipulation facilities. Goals may also be executed directly. The middle button menu contains the following entrees (see Figure 5.7):

Figure 5.7: A Dialog Window

```
again
undo
copy
cut
paste
parse it
prove it
```

Text to be used as input to the IG can be selected and parsed with the `parse it` entree. As the parsing proceeds, messages are written to the System Transcript. For example, parsing the following text:

```
Maier gave Smith a B in cs101.
```

Transforms the input into a ψ-query which is then evaluated. This grammar uses the built-in predicate `write` to add the following text to the dialog window:

```
>>> Okay.
>>> maier gave smith a b in cs101.
```

The first line indicates that the grammar has added this fact to its knowledge base; this is done by clauses in the semantic processing page. The second line is a translation from the internal representation back into English; see the generation grammar page. The query succeeds and a prompter appears, asking whether backtracking is desired. Since there are no other solutions for this example query, `no` may be selected. When `yes` is selected, the interpreter will backtrack. In any case, the interpreter then displays some timing information on the System Transcript, including total elapsed time (see Figure 5.8).

The clauses performing the semantic processing use the built-in predicate `assert` to add new clauses to the grammar. If the `Knowledge Base` page is viewed with a rule browser (see Figure 5.9), then the new clause representing the asserted information is seen.

System Transcript

```
Goal To Prove:
s(LogicTranslation,["maier","gave","smith","a","b","in","cs101"],[]).
HERE IS A SOLUTION (5343 msec):
LogicTranslation = take(smith; cs101; maier; b)
    TIME SPENT COMPUTING MAYBE-SETS = 173 msec.
    TIME SPENT UNIFYING = 1869 msec.
    TIME SPENT DOING GLBs = 574 msec.
    TOTAL ELAPSED TIME = 9946 msec.
```

Rule Browser

```
/***** ...ntax section **********/
s(
s(
vp
        vpComplements(Complements).
vpComplements([np(Complement)|Complements]) -->
        np(Complement),vpComplements(Complements).
vpComplements([]) --> .
vpComplements([in(Complement)|Complements]) -->
        [in],np(Complement),vpComplements(Complements).
```

Dialog

```
Maier gave Smith a B in CS101.
>>> Okay.
>>> maier gave smith a b in cs101.
```

Figure 5.8: Parsing a Statement

IG

Inheritance Grammar
Harry Porter

System Transcript

```
Goal To Prove:
s(LogicTranslation,["maier","gave","smith","a","b","in","cs101"],[]).
HERE IS A SOLUTION (5343 msec):
LogicTranslation = take(smith; cs101; maier; b)
    TIME SPENT COMPUTING MAYBE-SETS = 173 msec.
    TIME SPENT UNIFYING = 1869 msec.
    TIME SPENT DOING GLBs = 574 msec.
    TOTAL ELAPSED TIME = 9946 msec.
```

Rule Browser

```
/********** knowledge base **********/

knowledge(take(smith,cs100,kieburtz,a)).
knowledge(take(jones,cs100,kieburtz,c)).
knowledge(take(jones,cs100,maier,d)).
knowledge(take(smith; cs101; maier; b)).
```

Figure 5.9: Viewing an Asserted Clause

If the following input is now parsed:

```
Did Maier give Smith a B in cs101?
```

the grammar will write

```
>>> Yes.
```

into the dialog window. See Figure 5.10.

The `prove it` entree in the dialog window can be used to execute ψ-queries directly, which is useful in debugging a grammar. The `parse it` entree performs some lexical analysis on the input and packages the result into a legal ψ-query. For example, parsing

```
Did Maier give Smith a B in cs101?
```

results in evaluating the query

```
:- s(LogicTranslation,
     ["did","maier","give","smith","a","b","in","cs101"],
     []).
```

The `prove it` option evaluates the text as a ψ-query, without performing the lexical transformation.

The final option on the main interpreter menu is `open signature window`, which opens a graphical view of the signature of the grammar. When this entree is selected, a window like that shown in Figure 5.11 pops up. The other window shows the corresponding ordering statements.

Also shown is the middle button menu associated with the view, which includes the following entrees:

System Transcript

```
Goal To Prove:
s(LogicTranslation,["did","maier","give","smith","a","b","in","cs101"
],[]).
HERE IS A SOLUTION (2516 msec):
LogicTranslation = take(smith; cs101; maier; b)
    TIME SPENT COMPUTING MAYBE-SETS = 175 msec.
    TIME SPENT UNIFYING = 1141 msec.
    TIME SPENT DOING GLBs = 263 msec.
    TOTAL ELAPSED TIME = 3681 msec.
```

Rule Browser

```
/*****    nowledge base  **********/
kr
kr
kr
kr
```

Dialog

```
Maier gave Smith a B in CS101.
>>> Okay.
>>> maier gave smith a b in cs101.
Did Maier give Smith a B in cs101?
>>> Yes.
```

Figure 5.10: Parsing a Question

add symbol

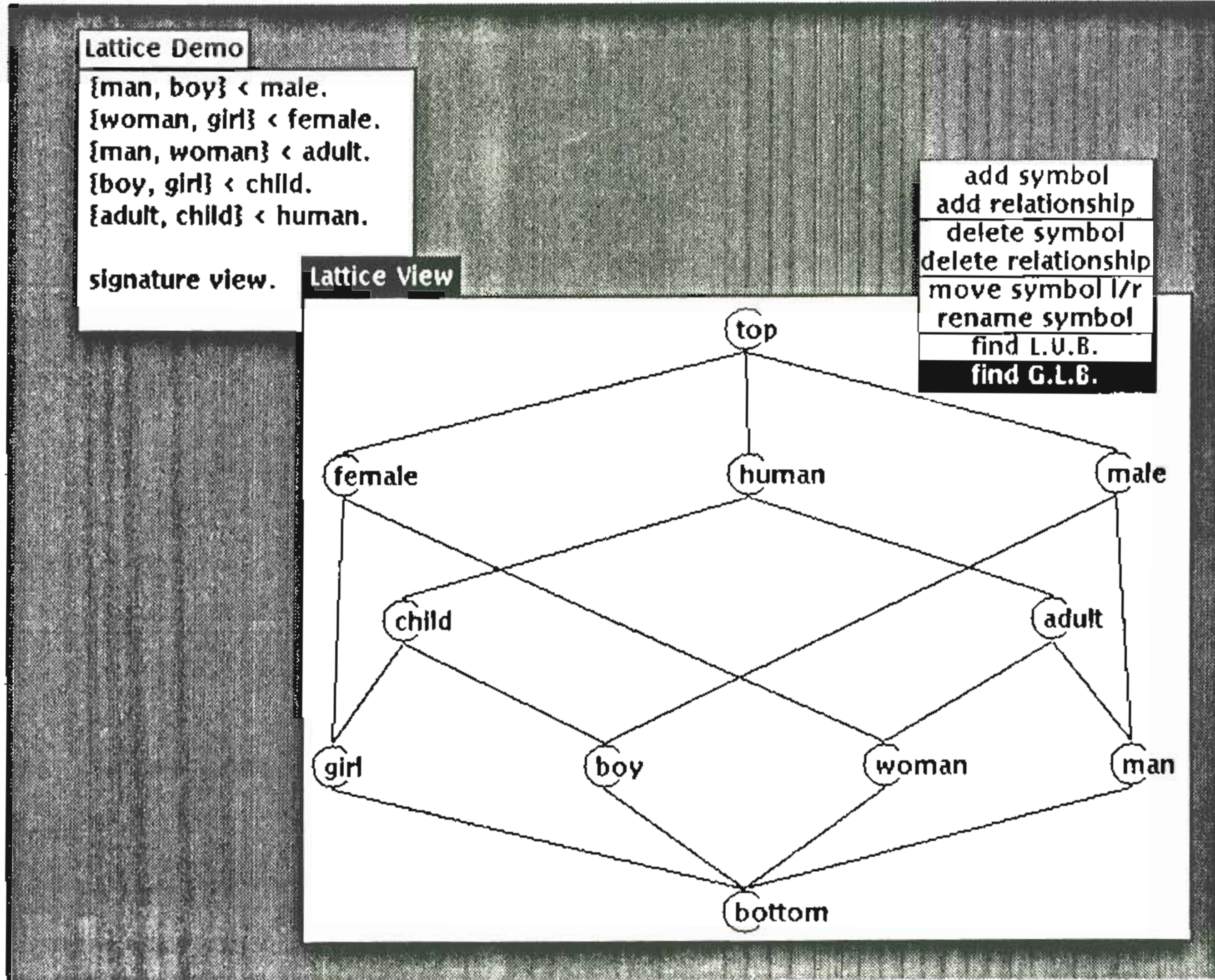

Figure 5.11: Editing a Signature

```
add relationship
delete symbol
delete relationship
move symbol l/r
rename symbol
find L.U.B.
find G.L.B.
```

When the last entree is selected, the system will wait for two elements in the order to be selected. Consider the elements `human` and `male`. In the ordering as it stands, both `boy` and `man` are lower bounds for these two elements – a unique GLB does not exist. The system will automatically create a new element and add it to the ordering. The view will then flash the new element, indicating that it is the GLB desired. The result is shown in Figure 5.12.

The first menu entree, allowing a symbol to be added, prompts the user to enter a name for the new symbol. The second entree allows new relationships to be added. The user must first select the greater symbol and then select the lesser symbol. (Only legal relationships may be added, i.e., cycles cannot be created.) Symbols and relationships may also be deleted from the ordering with the second and third entrees. The lattice package attempts to display the lattice in the most aesthetic way possible, but sometimes it fails. The fifth entree allows symbols to be moved around on the view. The sixth entree allows symbol names to be changed. For example, the symbol created by the GLB command is automatically given the name `{boy,man}`. This entree might be used to change that name to `humanMale`. Finally, the `find L.U.B.` entree works similarly to the `find G.L.B.` entree, described above.

5.5. Implementation-Specific Grammar Details

The interpreter accepts grammars whose forms vary slightly from *Inheritance Grammars* as described in Chapter 4. For example, built-in predicates and comments

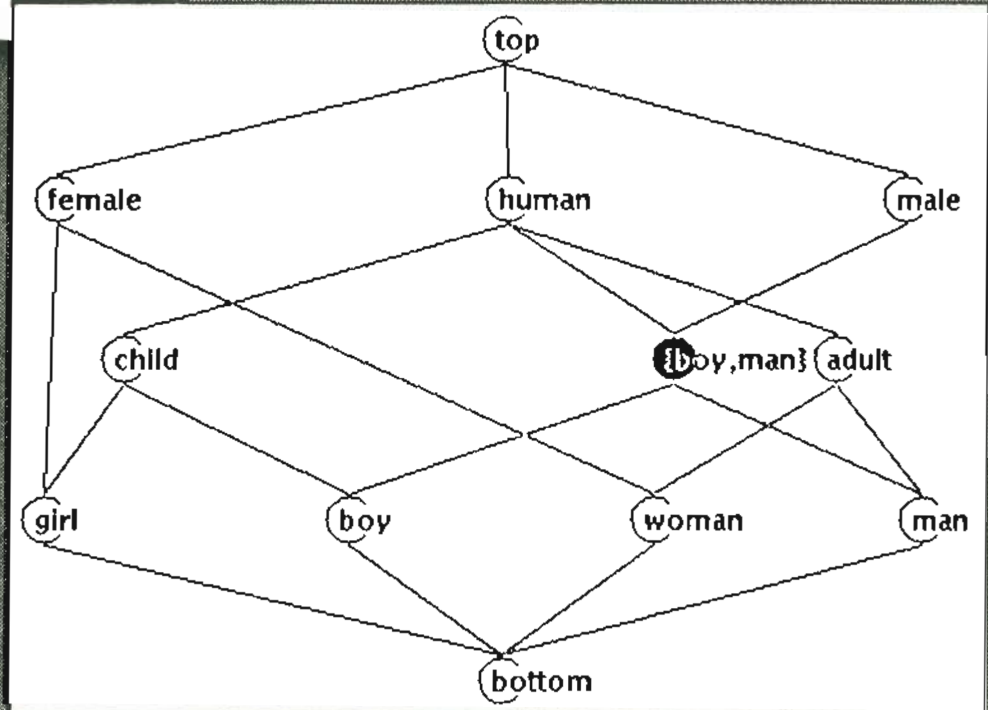

Figure 5.12: Finding a GLB

are allowed. Next, we discuss the form accepted by the interpreter. If the user attempts to `accept` a page that does not conform to this definition (e.g., the page contains a syntax error) a descriptive error message, indicating a location within the page, pops up.

When each page is `accept`-ed it is first processed by a lexical analyzer, which breaks the grammar into tokens. The tokens used here are the standard types of tokens that many programming language compilers recognize: identifiers, numbers, and punctuation symbols. White space (blanks, tabs, newlines, and comments) must be used to separate tokens that would otherwise be interpreted differently when concatenated. There are two notations for comments. A comment may begin with `/*` and run through the next `*/` (C syntax) or a comment may begin with `--` and run through end-of-line (Ada syntax). Both forms may be used in the same grammar.

Identifiers are used for ψ-term symbols, variables, and feature labels. An identifier must begin with an alphanumeric and may contain alphanumerics and hyphens. Arbitrary strings are accommodated as identifiers that do not conform to this constraint. To use such an identifier, it must be enclosed in quotes (either single or double). The delimiter may itself be included in the string by doubling it. An identifier consisting of a sequence of digits, possibly preceded by a hyphen (minus), is considered a *numeric* identifier and is treated specially by several built-in predicates, but not by the lexical analysis. Examples of legal identifiers are

```
noun-phrase
X
'o''clock'
"Ronald ""Ronnie"" Reagan"
-273
```

Figure 5.13 gives an extended BNF describing the exact syntax of the grammar accepted by the system. (As before, brackets enclose optional constituents, the vertical bar separates alternate constituents, and ellipses are used to indicate repetition.) The symbols

```
( ) { } < => = ¦ --> :- ; [ ] . , :
```

are terminals. In particular, note that we use `=>` for $\Rightarrow$ and note the difference between the terminals ¦ [ ] and the metasymbols | [ ]. With Chapter 4 as background, most of this syntax should be self-explanatory. A few comments are, however, in order.

Variable tags are identifiers that must begin with an uppercase letter. Predicates, labels, and symbols are identifiers that must not begin with an uppercase letter. Feature labels can be omitted for the first zero or more attributes in a term; in this case, feature labels are automatically generated from the integers `1`, `2`, `3`, ... For example,

```
np(third, singular, lex=>runs)
```

is the same as the term

```
np(1⇒third, 2⇒singular, lex⇒runs)
```

Variables are assumed to be quantified over clauses. Variables with the same spelling appearing in different clauses are assumed to be distinct. Within a clause, if several occurrences of one variable (say `X`) are followed by a colon and a term, all such terms are unified at parse time. If they cannot be unified an error message will be displayed.

Bracket notation can be used for lists of terms. A list is automatically translated, during parsing, into an equivalent ψ-term using the symbols `cons`, `head`, `tail`, and `[]`. For example, the input

```
GRAMMAR         ::=     PAGE PAGE ... PAGE
PAGE            ::=     STATEMENT STATEMENT ... STATEMENT
STATEMENT       ::=     CLAUSE | GRAMMARRULE | ORDERINGSTMT | EQDEF

CLAUSE          ::=     LITERAL :- BODY .
                    |   LITERAL .
BODY            ::=     LITERAL , LITERAL , ... LITERAL
LITERAL         ::=     PREDICATE
                    |   PREDICATE ( TERM , TERM , ... TERM )
PREDICATE       ::=     IDENT

TERM            ::=     SYMBOL [ ATTRLIST ]
                    |   ATTRLIST
                    |   VAR [ : TERM ]
                    |   TERMLIST
SYMBOL          ::=     IDENT
VAR             ::=     IDENT
ATTRLIST        ::=     ( ATTR SEPARATOR ATTR SEPARATOR ... ATTR )
SEPARATOR       ::=     , | ;
ATTR            ::=     FEATURE => TERM
                    |   TERM
FEATURE         ::=     IDENT
TERMLIST        ::=     [ TERM , TERM , ... TERM [ | TERM ] ]
                    |   [ ]

ORDERINGSTMT    ::=     SYMBOL < SYMBOL .
                    |   { SYMBOL , SYMBOL , ... SYMBOL } < SYMBOL .

GRAMMARRULE     ::=     LITERAL --> RULEBODY .
                    |   LITERAL --> .
RULEBODY        ::=     RULEITEM , RULEITEM , ... RULEITEM
RULEITEM        ::=     NONTERMINAL
                    |   TERMINALLIST
                    |   NONRULECODE
NONTERMINAL     ::=     LITERAL
TERMINALLIST    ::=     TERMLIST
NONRULECODE     ::=     { BODY }

EQDEF           ::=     SYMBOL = TERM .
```

Figure 5.13: Syntax Accepted by the Implementation

```
[a,b,c]
```

is parsed as the term

```
cons (head => a,
      tail => cons (head => b,
                    tail => cons (head => c,
                                  tail => []))))
```

and the input

```
[a, X, c | X]
```

is parsed as the term

```
cons (head => a,
      tail => cons (head => X,
                    tail => cons (head => c,
                                  tail => X)))
```

If the punctuation symbol `-->` is used in a clause in place of `:-` then the clause is assumed to be a IG *grammar rule*. That is, the clause follows a syntax similar to Definite Clause Grammar syntax. The body consists of some combination of non-terminal literals, terminals, and code. The head literal and the non-terminal literals in a grammar rule are automatically augmented (during parsing) by adding two arguments to facilitate depth-first parsing. (See the discussion of Definite Clause Grammars in Chapter 1.) These arguments are each a ψ-term consisting of only a variable and no head symbol or attributes. The variables names are generated automatically from identifiers such as `P1`, `P2`, `P3`, etc. Terminal symbols of the target language are indicated by enclosing them in brackets in the body of the grammar rule. The parser automatically generates a sequence of queries to pick off the terminals. Finally, if arbitrary goal literals are needed within the grammar rule and the grammar writer wishes to circumvent the automatic addition of the position arguments, these literals can be enclosed in curly braces.

These remarks are summarized in an example. Consider the following grammar rule:

```
s(likes(X,Y))  -->  properNoun(X),
                    [likes,a],
                    noun(Y),
                    {check([X,Y])}.
```

The meaning of this contrived rule might be that a sentence consists of a proper noun followed by two terminal words followed by a common noun. After parsing the common noun, we need to call a predicate (named `check`) to perform some semantic checking. We then succeed, returning a term through the clause head. The grammar rule notation is syntactic sugar accepted by the system, which translates this clause exactly the same way it would translate the following clause:

```
s(likes(1=>X,2=>Y),P1,P5)  :-
            properNoun(X,P1,P2),
            dcgConnect(likes,P2,P3),
            dcgConnect(a,P3,P4),
            noun(Y,P4,P5),
            check(cons(head=>X,
                       tail=>cons(head=>Y,
                                  tail=>[])))).
```

5.6. Built-In Predicates

The interpreter understands the following built-in predicates and handles them specially when they appear in goals. The predicates are listed with an indication of their arguments.

```
lessThan(X,Y)
greaterThan(X,Y)
lessThanOrEqual(X,Y)
greaterThanOrEqual(X,Y)
plus(X,Y,Z)
write(Term)
nl
assert(Clause)
call(Goal)
not(Goal)
```

All these predicates operate similarly to their counterparts in Prolog. The `write` and `nl` predicates send their output to the dialog window, where it is inserted after the input that was selected for the `parse it` menu entree.

Chapter 6: Additional Evaluation Strategies[1]

6.1. Introduction

In the Chapter 4, we gave a description of ψ-logic and the Inheritance Grammar formalism. We then provided a semantics for ψ-logic programs and introduced the concept of an evaluation (or proof) procedure for executing ψ-logic programs (i.e., for parsing with Inheritance Grammars). The traditional Prolog interpreter can be adapted to the execution of ψ-logic programs and, in Chapter 5, we discussed two implementations based on Prolog execution techniques. The traditional evaluation strategy, however, performs a depth-first search for a proof and consequently suffers from some of the problems associated with depth-first parsing algorithms. For example, if a program contains left-recursive rules the interpreter may go into an infinite loop, failing to find a proof even though one exists, just as a top-down parser may fail to find a parse when given a grammar containing left-recursion[2]. This type of behavior motivated the definition of completeness for evaluation strategies.

In this chapter, we discuss several alternative execution strategies developed for executing Horn-clause logic programs. While not originally intended for ψ-logic, we describe how they can be applied to IG evaluation. These evaluation strategies are algorithms for scheduling unifications and manipulating clauses; they do not depend on

[1] Portions of this chapter appeared in [Porter 1986].

[2] We want to be able to express the grammar as clearly as possible, without letting implementation details like clause/literal order get in the way. As in many logic programs, the rules in logic grammars express general knowledge about language and it is often difficult to foresee how they will be used.

particulars of the entities being unified. In general outline, an evaluation strategy designed for Horn-clause logic using first-order terms can be applied to the evaluation of ψ-logic queries by replacing the concepts of *term*, *unification*, and *resolution* where they occur in the description of the strategy by *ψ-term*, *ψ-term unification*, and *ψ-term resolution* respectively.

The first strategy we discuss is *Earley Deduction*, a generalization of the Earley Parsing Algorithm [Earley 1970] to the execution of logic programs. For first-order logic, we will show that Earley Deduction is both complete and sound. Then, restricting our attention to a special case of first-order terms – namely functor-free programs – we show that Earley Deduction is guaranteed to terminate[3]. We implemented the Earley Deduction algorithm (for first-order terms, only) and summarize the techniques used and results obtained. Earley Deduction is elegant but, unfortunately, not very (space) efficient. We investigated implementation techniques that may make Earley Deduction practical for Datalog programs and these are also described.

The second strategy is the *extension tables algorithm* [Dietrich and Warren 1986] and can be viewed as an adaptation of *memo tables* (for saving results in functional programming) to the domain of logic programs. While not as elegant as Earley Deduction, the method offers some flexibility that may make it more useful in executing IGs.

The final strategy is called Staged Depth-First Search Strategy [Stickel 1984]. This simple technique just bounds the search space. By repeatedly increasing this bound after the search space has been exhaustively searched, a complete search strategy

[3] The functor-free subset of Prolog is called Datalog and can be used to compute relational join, selection, transitive closure, etc. The discussion of Datalog is not directly pertinent to natural language parsing, but is indirectly relevant since the primary application envisioned for IGs is for front-end access to deductive relational databases.

obtains. We conclude the chapter with a discussion of the application of the various search strategies to the execution of Inheritance Grammars.

6.2. Earley Deduction

We begin this subsection by describing Earley Deduction via tracing its execution on a small example program. Our strategy is to first describe the algorithm using a first-order logic example and then to describe the application of the algorithm to ψ-logic programs. We then discuss correctness and completeness of the algorithm and show that it terminates for functor-free programs. Finally, we discuss implementation techniques and give results for a version of the algorithm restricted to functor-free programs.

6.2.1. Description of Earley Deduction

In 1970, Jay Earley described an algorithm called *Earley Parsing*, which overcomes some of the problems of both top-down and bottom-up parsing by combining aspects of both [Earley 1970]. In computational linguistics, various incarnations of this algorithm have been studied under the name *Active Chart Parsing* [Kaplan 1973]. F. C. N. Pereira and D. H. D. Warren extended Earley Parsing to the execution of logic programs and call the method *Earley Deduction* [Pereira and Warren 1983]. Their algorithm has three desirable properties. First, it is sound and complete. Second, it is guaranteed to terminate for an interesting subset of logic, namely functor-free programs. Finally, Earley Deduction is straightforward and easy to implement.

To illustrate the algorithm, we use an example program containing a transitive closure rule (which would cause nontermination for a depth-first evaluation strategy). The clauses comprising the example are:

```
p(X,Z)  :- p(X,Y), p(Y,Z).          (1)
p(a,b).                             (2)
p(b,c).                             (3)
```

These are called the *program clauses*. (As usual, variables are assumed to be quantified over individual clauses: variables in different clauses are assumed to be distinct.)

The goal is transformed into a *goal clause* by adding a dummy literal (with head symbol `ans`) as the clause head. The attributes of this `ans` literal will be used to accumulate the bindings computed in a successful proof. The goal clause we will use is:

```
ans(Z)  :- p(a,Z).                  (4)
```

This goal has only one literal but, in general, there will be several literals on the right-hand side.

The method works by building up a set of *derived clauses.* As an initialization step, the *goal clause* is added as the first element to the set of derived clauses. Each step of the method adds another clause to the set of derived clauses and, when no more clauses can be added, terminates.

There are two inference rules, called *reduction* and *instantiation*. Both rules work by combining a derived clause with another clause (either program or derived). The former (derived) clause is called the *selected clause* and the latter clause is called the *side clause.* One literal within the body of each derived clause will be marked as the *selected literal.* It is chosen when the clause is first created and added to the derived clause set. The choice is arbitrary; we will always select the left-most literal in the body of a derived clause[4].

---

[4] Choosing the left-most literal gives the top-down aspect of the algorithm a left-to-right orientation; choosing the right-most literal would give a right-to-left orientation.

The reduction step applies when the side clause is a unit clause and is a special case of the regular resolution rule. Reduction works as follows. First, the selected literal of the selected clause is unified with the unit clause. The selected clause must always be a derived clause, but the side clause can be either a program or a derived clause. Let σ be the most general unifier of the selected literal and the head of the unit clause. Second, remove the selected literal from the selected clause, apply σ to what remains and add the result as a new derived clause.

For example, let clause (4) be the selected clause and let clause (2) be the side clause. The selected literal of the selected clause is `p(a,Z)` since it is the left-most literal on the right-hand side. The unifier is $\sigma = \{ Z \leftarrow b \}$. Removing the selected literal gives `ans(Z)` and applying the unifier gives `ans(b)`, which is added to the derived clause set.

```
ans(b).                                                    (5)
```

Whenever a unit clause is derived that has `ans` as its head symbol, it is output as a solution. Thus, `ans(b)` is printed as an answer. Earley Deduction never backtracks: it continues executing, producing answers along the way, until it terminates (if ever).

The second rule is *instantiation*. For this rule, we take the selected literal of the selected clause and unify it with the positive literal (i.e., the head) of a non-unit program clause (the side clause), giving a most general unifier σ. We then apply σ to the program clause and add the result as a new derived clause.

To illustrate this rule, we use clause (4) as the selected clause to instantiate clause (1). The unification of the selected literal `p(a,Z)` with the head of clause (1) `p(X,Z)` gives $\sigma = \{ X \leftarrow a \}$. (Technically, the two `Z`'s should be renamed to distinct names and σ should then bind these names together. In this example the result is the same.)

Instantiating the program clause with σ gives the new derived clause.

```
p(a,Z) :- p(a,Y), p(Y,Z).                                        (6)
```

Continuing the inferencing, the reduction rule can be applied to clause (2) and clause (6). We say that clause (2) *reduces* clause (6) and the result is clause (7):

```
p(a,Z) :- p(b,Z).                                                (7)
```

It is possible for a reduction or instantiation step to produce a clause that has been derived earlier. For example, clause (6) can now be used to instantiate clause (1) but the result has already been derived (as clause (6) itself). To avoid this redundancy, we stipulate that a clause is not to be added as a new derived clause if it is subsumed by an already-derived clause.

For our purposes, we will say that a first-order clause

$$p_0 \text{ :- } p_1, \ldots, p_j.$$

subsumes another clause

$$q_0 \text{ :- } q_1, \ldots, q_k.$$

when j=k and when every literal p_i subsumes the corresponding literal q_i. The obvious way to perform this check is to take a new candidate clause and look through all the derived clauses, performing the subsumption check on each. This blind searching can be quite time consuming and we will have something to say below about doing it more intelligently.

We complete the specification of Earley Deduction by specifying how the algorithm selects pairs of clauses for combination. Not every selection strategy will find answers even when they exist, as the sequence in Figure 6.1 demonstrates.

```
Program Clauses:
   p(X) :- p(f(X)).                                          (14)
   p(a).                                                     (15)
Derived Clauses:
   ans :- p(a).                     Goal                     (16)
   p(a) :- p(f(a)).                 16 instantiates 14       (17)
   p(f(a)) :- p(f(f(a))).           17 instantiates 14       (18)
   p(f(f(a))) :- p(f(f(f(a)))).     18 instantiates 14       (19)
    •
    •
    •
```

Figure 6.1: An Infinite Deduction Sequence

At some step in the algorithm let clauses C_1 and C_2 be two clauses that can be combined (either using instantiation or reduction) to produce a new clause C_3. We require a selection strategy to eventually get around to combining C_1 and C_2 and considering C_3. Consequently, a clause at least as general as C_3 will eventually be added to the derived set, since C_3 itself will be added unless it is subsumed by some other previously derived clause. We call such a selection strategy *fair*.

Assume the program clauses are numbered from 1 to n, clause n+1 is the goal clause and new derived clauses are numbered sequentially from n+2 as they are added. Here is the obvious fair scheduling policy:

```
i := n+1
repeat
    for j := 1 to i-1 do
        Attempt to combine clause i and clause j using
        instantiation and reduction and add any new
        clauses to the set of derived clauses.
    endfor
    i := i+1
until i > the number of clauses
```

In our transitive closure example, there are several more instantiations and reductions we can perform before we reach a point where no new clauses can be derived. The deduction quickly terminates and we show the resulting clauses in Figure 6.2. We have included comments and, for convenience, clauses (1) through (7) are repeated. (Note that, although a fair scheduling policy was used, it was not the one listed above.)

6.2.1.1. Discussion

In examining these clauses, note how the right-hand sides of derived non-unit clauses represent goals that need to be solved to produce an answer. For example, the right-hand side of clause (7) indicates that we need to solve `p(b,Z)` in order to complete a proof. The head of clause (7) is the subgoal that motivates the proof of `p(b,Z)`, namely `p(a,Z)`. The goal `p(b,Z)` was encountered while trying to solve the body of clause (6) and was produced when clause (2) reduced clause (6).

```
Program Clauses:
    p(X,Z)  :- p(X,Y), p(Y,Z).                                  (1)
    p(a,b).                                                     (2)
    p(b,c).                                                     (3)
Derived Clauses:
    ans(Z)  :- p(a,Z).              Goal                        (4)
    ans(b).                         2 reduces 4                 (5)
    p(a,Z)  :- p(a,Y), p(Y,Z).      4 instantiates 1            (6)
    p(a,Z)  :- p(b,Z).              2 reduces 6                 (7)
    p(b,Z)  :- p(b,Y), p(Y,Z).      7 instantiates 1            (8)
    p(a,c).                         3 reduces 7                 (9)
    p(b,Z)  :- p(c,Z).              3 reduces 8                 (10)
    p(c,Z)  :- p(c,Y), p(Y,Z).      10 instantiates 1           (11)
    ans(c).                         9 reduces 4                 (12)
    p(a,Z)  :- p(c,Z).              9 reduces 6                 (13)
```

Figure 6.2: A Completed Earley Deduction

The algorithm has a top-down component since new goals are only produced when needed to satisfy existing goals. The instantiation rule provides the top-down behavior: the selected literal of a clause body drives instantiation, which recruits a program rule to solve the literal. Thus, goals are only created when their solution is relevant in solving the goal clause.

The algorithm also has a strong bottom-up flavor since once proven, solutions are stored and reused, rather than recomputed. The bottom-up behavior comes from the reduction rule: once a unit clause is derived, it can be used to satisfy selected literals in any number of clause bodies. For example, once a solution for `p(a,Z)` is produced (e.g., clause (9)) it can be used by both clause (4) and clause (6).

6.2.2. Application of Earley Deduction to ψ-Logic

The Earley Deduction algorithm can be applied to the execution of ψ-logic programs by providing definitions of the reduction and instantiation rules for ψ-clauses. Adopting the notational conventions of Chapter 4, we assume that the selected clause C_1 is a quintuple $<\mathbf{S}_1, s_1, \mathbf{A}_1, \psi_1, \tau_1>$ and is represented schematically as

$$C_1:\ t_0 :- t_1, \ldots, t_i, \ldots, t_j.$$

where $1 \leq i \leq j$. Literal t_i is the selected literal. When the selected literal is always the left-most literal of the selected clause's body, we have $i = 1$.

The side clause C_2 is a quintuple $<\mathbf{S}_2, s_2, \mathbf{A}_2, \psi_2, \tau_2>$ and is represented as

$$C_2:\ t_{j+1} :- t_{j+2}, \ldots, t_k.$$

where $k \geq j+1$.

Reduction of ψ-clauses is a special case of ψ-resolution, namely when the side clause C_2 has no body, i.e., $k = j+1$. In such a case, we represent the side clause as

C_2: t_{j+1}.

The instantiation of program clause C_2 by a side clause C_1 requires that C_2 be a rule, i.e., that $k > j+1$. The result of the instantiation can be given in the notation of Chapter 4 as

$$(C_2 \sqcap (C_1 \{ 0 \leftarrow 0, \ldots, i\text{-}1 \leftarrow i\text{-}1, j+1 \leftarrow i, i+1 \leftarrow i+1, \ldots, j \leftarrow j\}))$$
$$\{\bullet \leftarrow 0, \ldots, \bullet \leftarrow i\text{-}1, \bullet \leftarrow i+1, \ldots, \bullet \leftarrow j, j+1 \leftarrow j+1, \ldots, k \leftarrow k\}$$

To illustrate Earley Deduction on a ψ-logic program, we have transformed our first-order example into a similar ψ-logic example, shown in Figure 6.3. The first program clause may be thought of as a simple grammar rule

```
adj --> adj, adj.
```

for parsing sequences of adjectives. Note how the entire query is collected and saved in the `ans` goal clause. When there is more than one literal in the query

:- $t_1, \ldots, t_j$.

we can use a list in the constructed goal clause:

ans(query$\Rightarrow$[X_1:t_1, ... , X_j:t_j]) :- X_1:t_1, ... , X_j:t_j.

6.2.3. Soundness of Earley Deduction

The Earley Deduction proof procedure is correct (*sound*) for both first-order logic and ψ-logic, in the sense that any answer obtained implies the query is a logical consequence of the program clauses. To show this, we first note that both inference rules are sound (see below). Then, by induction on the order of the derived clauses, each derived clause is a logical consequence of the program clauses and goal clause.

```
Program Clauses:
   adj(from⇒P1, to⇒P3) :-
         adj(from⇒P1, to⇒P2), adj(from⇒P2, to⇒P3).                 (1)
   adj(from⇒1, to⇒2).                                               (2)
   adj(from⇒2, to⇒3).                                               (3)
Derived Clauses:
   ans(query⇒Q) :- Q:adj(from⇒1).        Goal                       (4)
   ans(query⇒adj(from⇒1, to⇒2)).         2 reduces 4                (5)
   adj(from⇒1, to⇒P3) :-
         adj(from⇒1, to⇒P2),
         adj(from⇒P2, to⇒P3).            4 instantiates 1           (6)
   adj(from⇒1, to⇒P3) :-
         adj(from⇒2, to⇒P3).             2 reduces 6                (7)
   adj(from⇒2, to⇒P3) :-
         adj(from⇒2, to⇒P2),
         adj(from⇒P2, to⇒P3).            7 instantiates 1           (8)
   adj(from⇒1, to⇒3).                    3 reduces 7                (9)
   adj(from⇒2, to⇒P3) :-
         adj(from⇒3, to⇒P3).             3 reduces 8                (10)
   adj(from⇒3, to⇒P3) :-
         adj(from⇒3, to⇒P2),
         adj(from⇒P2, to⇒P3).            10 instantiates 1          (11)
   ans(query⇒adj(from⇒1, to⇒3)).         9 reduces 4                (12)
   adj(from⇒1, to⇒P3) :-
         adj(from⇒3, to⇒P3).             9 reduces 6                (13)
```

Figure 6.3: Earley Deduction using ψ-Logic

Recall that a first-order *definite clause* is a disjunction of literals, one positive and zero or more negative, although it is more intuitively written using an implication whose antecedent is a conjunction of positive literals. The query, a conjunction of positive literals, is negated (and then re-written as a disjunction of negative literals) and the proof is by refutation. The contradiction is represented by the *empty clause*. In the Earley Deduction algorithm, the unit clause `ans(...)` denotes the empty clause and also captures information about the bindings obtained in its derivation. The technique of adding additional dummy literals to clauses to accumulate answer substitutions is well-known (e.g. [Nilsson 1971]).

In a traditional refutation proof, there is only one inference rule: resolution. Here we have 2 rules, instantiation and reduction. Reduction is clearly a special case of resolution, namely when one of the two clauses consists of a single positive literal. Thus, by Theorem 8 of Chapter 4, the reduction rule is sound for the case of ψ-logic. A clause produced by instantiation can also be seen to be a logical consequence of previous clauses, since it is just an instantiated version of an axiom. (Theorem 7 of Chapter 4 showed this result for ψ-logic.) Thus, if the unit clause `ans(...)` is derived, the empty clause has been produced and thus the refutation has been shown. Figure 6.4, which shows graphically the relationships between derived clauses in a proof of `ans(c)` from the earlier first-order example in Figure 6.2, may make the correspondence between Earley Deduction proofs and the resolution proof process clearer.

6.2.4. Completeness of Earley Deduction

In this section we provide a proof that Earley Deduction is a complete evaluation strategy for first-order programs. We conjecture that Earley Deduction is also complete for ψ-program execution. We begin with the definition of *Earley Deduction Trees.*

Definition. Given an Earley Deduction, an *Earley Deduction Tree* is a tree in which every node is labeled with a clause from the deduction and in which the root is labeled with an answer clause

```
ans  ( ... )
```

that has no body. The root and each interior node corresponds to one application of the reduction or instantiation rule and has two children, corresponding to the selected and side clauses used in that rule application. A leaf is either a program clause, the goal clause, or a derived unit clause and the clause corresponding to every parent will have been derived after the clauses corresponding to its children in the Earley

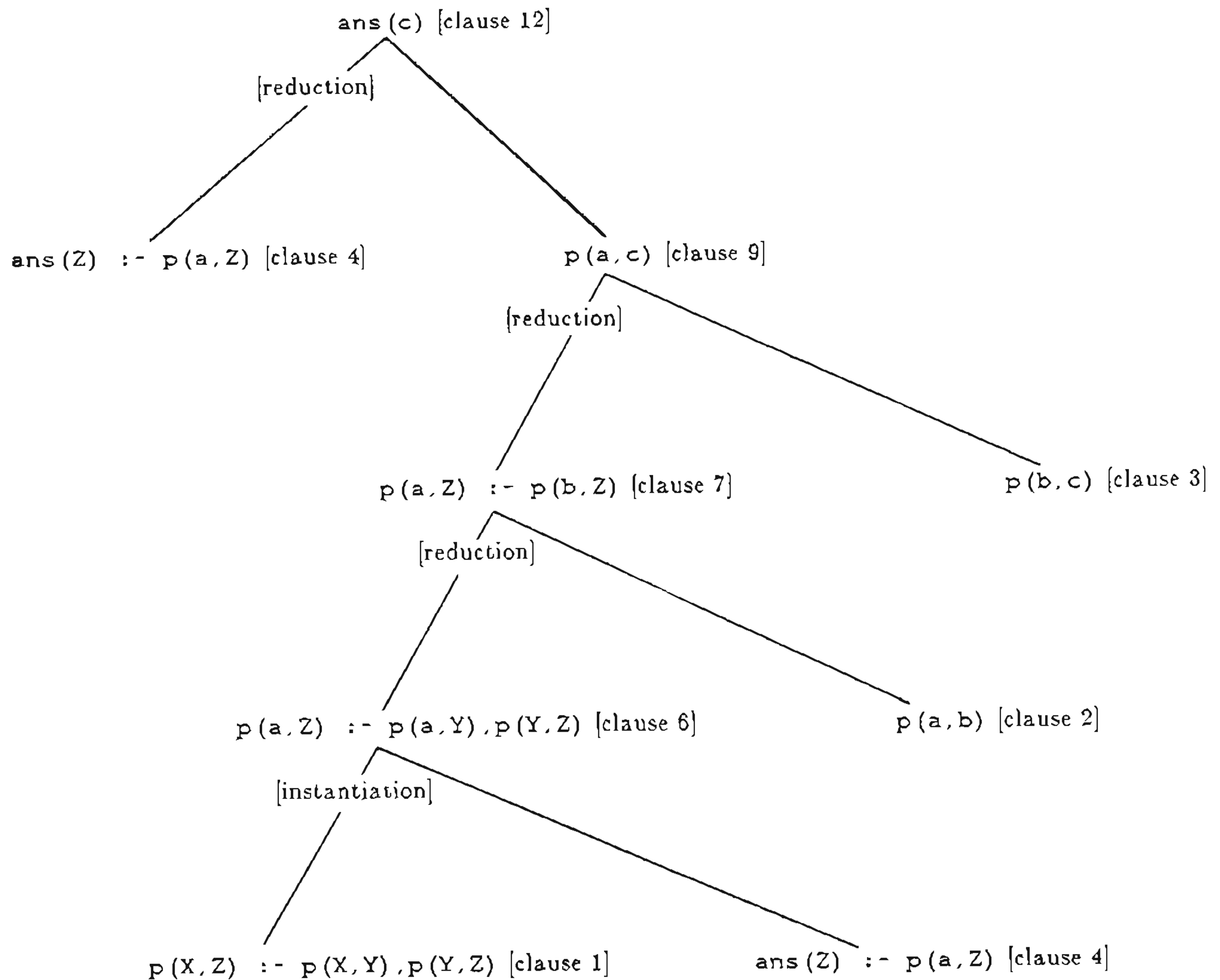

Figure 6.4: An Earley Deduction Tree

Deduction. □

For clauses produced by reduction, one child will be a unit clause, since one of the clauses used in the reduction step must be a unit clause. Clauses produced by

instantiation will simply be less general instantiations of some program rule. Such an Earley Deduction tree clearly exists for every answer produced by the Earley Deduction.

Example. Figure 6.4 shows an example Earley Deduction tree based on the deduction given in Figure 6.2. Comments are shown in brackets and, technically, are not part of the tree. □

Since it should be obvious which child is the selected clause and which is the side clause, the order of the two children in the graphical presentation of Earley Deduction tree is unimportant.

Definition. A *proof tree* is a tree in which each node is labeled with a clause from the program, perhaps with a substitution applied. The root node of the proof tree is the query clause with the answer substitution applied. Each child of an interior node is associated with one literal in the body of its parent and each parent has one child for each literal in its body. Consequently, every leaf is labeled with a unit clause and every interior node is labeled with a non-unit clause. The head literal of a child matches exactly the corresponding literal in the body of its parent. (Note that some authors attach a substitution to each node instead of performing the substitution on the clause.) □

Example. Using the program given in Figure 6.2, Figure 6.5 shows an example proof tree for the answer

```
p(a,c).
```

□

The proof tree corresponds closely to a resolution-based proof. If a resolution-based proof of a query Q exists, then such a proof tree with root labeled Q can be constructed.

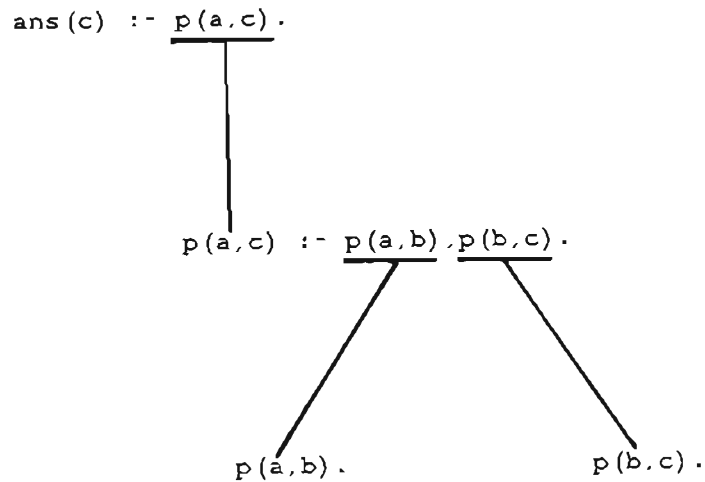

Figure 6.5: A Proof Tree

In showing that Earley Deduction is complete, we will appeal to the (refutation) completeness of binary resolution for Horn clauses. Since we have not shown extended this result to to ψ-logic, the completeness result for Earley Deduction using ψ-logic must be qualified with the words *Assuming binary resolution to be refutation complete for ψ-clauses...*

After making this assumption, we must show that every answer that can be produced by any refutation proof strategy will be produced by Earley Deduction. Our approach is to consider one such answer solution — that is, to consider a query that logically follows from the program clauses. Since there is a straightforward correspondence between proofs and proof trees, if the query is true, then a proof tree exists, although not all proof strategies will necessarily discover it. We show how this proof tree can be used to construct an Earley Deduction tree. Finally, we show that Earley Deduction

will produce a proof of the answer solution that corresponds to an Earley Deduction tree at least as general as the constructed tree.

We begin by showing how to construct an Earley Deduction tree from a proof tree. Call the result the *constructed tree.* In the construction, we will associate an Earley *tree* with every *node* in the proof tree. The construction is defined recursively, starting at the leaves of the proof tree and working toward the root. The Earley tree associated with the root is the resulting constructed tree.

The leaves of the proof tree are unit clauses from the program. With these, associate one-node Earley trees consisting of the same program unit clauses.

The translation of interior nodes of the proof trees is shown diagrammatically in Figures 6.6 and 6.7. Figure 6.6 shows an interior node of the proof tree, labeled

$$p() \text{ :- } q_1(), q_2(), \ldots, q_n()$$

with n children. The subtrees labeled $q_i()$ represent the *Earley Deduction trees* associated with each of the interior node's n children. These Earley trees are labeled with the

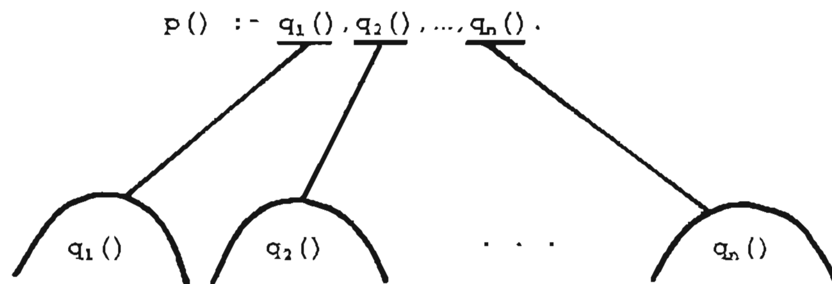

Figure 6.6: An Interior Node

head literal of the root node of the ith tree, namely `qi()`. Figure 6.7 shows the Earley tree to be associated with the interior node of the proof tree shown in Figure 6.6.

As an example, Figure 6.8 shows the constructed tree associated with the proof tree shown in Figure 6.5. All interior nodes in the constructed Earley Deduction tree correspond to reduction steps. The root of the proof tree will always be labeled with a clause with the `ans()` predicate as its positive literal and the root of the constructed tree will always be labeled with a unit `ans()` clause. In Figure 6.8, the root is labeled `ans(c)`.

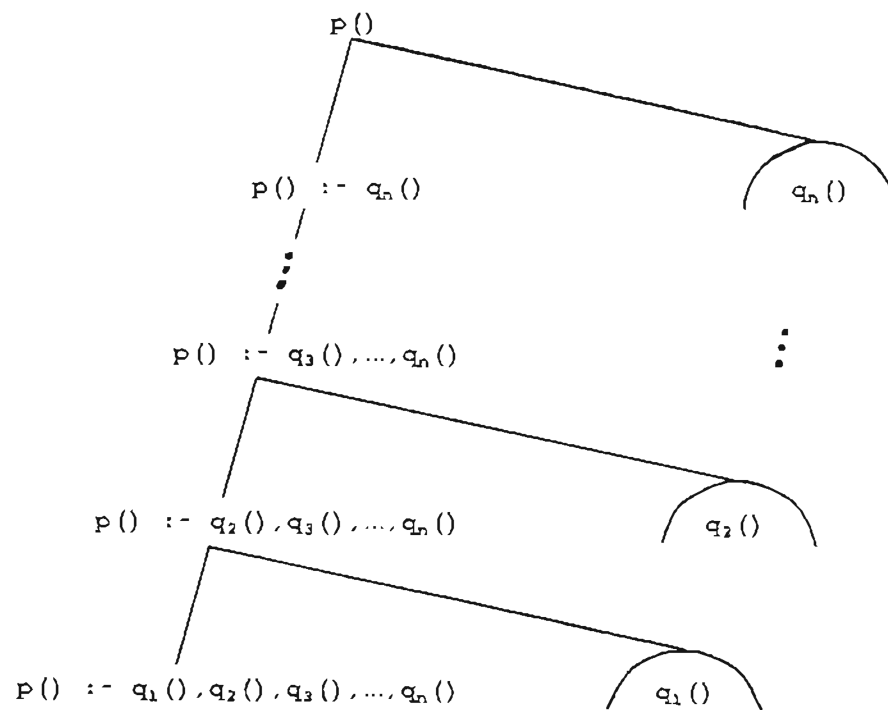

Figure 6.7: Resulting Earley Deduction Tree

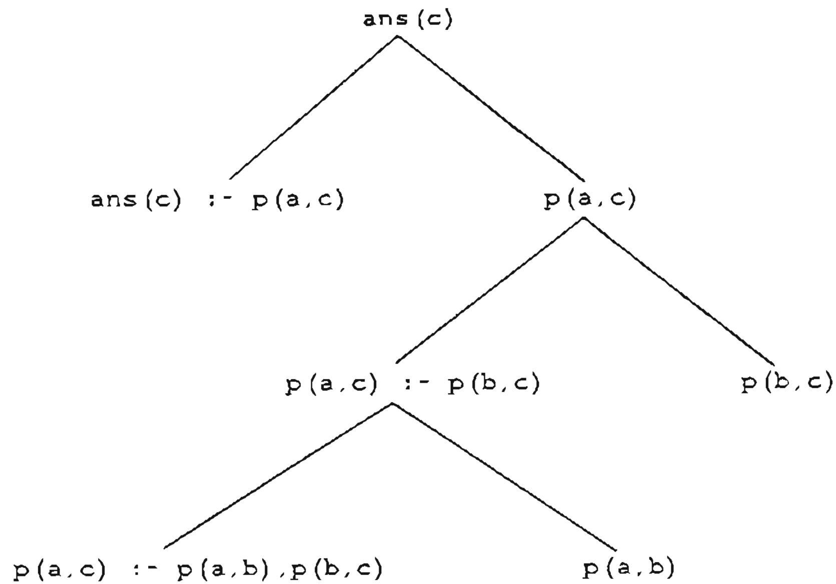

Figure 6.8: A Constructed Earley Deduction Tree

To conclude our proof, we must show that, given such a constructed Earley Deduction tree, Earley Deduction will produce an answer at least a general as the answer labeling the root of the tree. The proof is by structural induction on the constructed Earley Deduction tree.

As the basis, note that the Earley Deduction algorithm begins with a collection of program and goal clauses containing clauses at least as general as each of the clauses labeling the leaves of the constructed tree. For the inductive step, consider an interior node of the constructed tree. It is labeled with a clause C_3 and has two children whose roots are labeled with clauses C_1 and C_2. If the Earley Deduction algorithm has already produced two clauses that are at least as general as C_1 and C_2 then, because

the selection strategy must be fair, these clauses will ultimately be combined to derive a clause at least as general as C_3.

Thus, we conclude that the algorithm will ultimately derive a clause which is at least as general as the head of the goal clause labeling the root of the constructed tree. Earley Deduction will ultimately derive the `ans()` clause (in this example, `ans(c)`) and, consequently, is a complete evaluation strategy, assuming that binary resolution itself is complete.

6.2.5. Termination for Datalog

We next restrict our attention to a special case of Earley Deduction. A *Datalog* program is a definite clause program using first-order terms that contain no functors. The example program from Figure 6.2 happens to meet the Datalog restriction. The functor-free subset of first-order logic is important because it can be used to express data and queries from relational algebra in a clear and concise way. In addition, it can be used to express queries involving recursively defined data (e.g., transitive closure queries), which are inexpressible in relational algebra.

Depth-first evaluation is unsatisfactory for executing such queries because left-recursive rules may cause non-termination and avoiding left-recursive rules may be inconvenient. Fortunately, Earley Deduction is guaranteed to terminate whenever the program clauses contain no functors.

Theorem. Earley Deduction always terminates when applied to a Datalog program. □

Proof. Every step of the deduction adds a new clause to the set of derived

clauses but no added clause is ever any longer than the longest program clause[5]. To see that infinitely long clauses can never be derived, consider the reduction and instantiation rules. Reduction takes a given clause (the selected clause) and removes the selected literal. Thus, reduction can't be used to make longer clauses. Instantiation takes a program clause and instantiates it. Since a variable can only be replaced with a single symbol, instatiation can not be used to make a longer clause either.

There are a finite number of predicates and constant symbols appearing in the program and goal clauses (assuming the Datalog program is finite in size, of course). Given a finite number of symbols and some size k, there are only a finite number of clauses of length k or fewer symbols. (Two clauses differing only in the names of variables are considered equal, so an infinite supply of variable names does not affect this bound.)

At any step in the deduction, there are only finitely many pairs of derived and program clauses and there are only two ways to combine each pair to produce a new clause. Given a newly produced clause, the subsumption check can also be done in a finite amount of time. Thus, there is either another clause that can be derived – in which case the algorithm will find it and add it to the derived set in finite time – or there are no new clauses that can be derived that have not already been added to the derived set – in which case the algorithm will halt.

Thus, since we are guaranteed to either produce a new derived clause in finite time or halt and since we are guaranteed to never produce clauses longer than a given bound and since there are only a finite number of such clauses, the process is guaranteed

[5] By *length* we mean number of symbols (i.e., lexical tokens), ignoring the number of characters in any given symbol.

to terminate. □

In the case of first-order terms in general, the derived clauses may be larger than either of the two existing clauses. For example, when we use the clause

```
p(a) :- p(f(a)).
```

to instantiate the clause

```
p(X) :- p(f(X)).
```

the result

```
p(f(a)) :- p(f(f(a))).
```

is larger than either original clause. Thus, the procedure is not guaranteed to terminate for programs not meeting the Datalog restriction.

6.2.6. Implementation

We produced two implementations to evaluate the Earley Deduction algorithm and explore several optimizations, both for first-order programs and for Datalog programs. The first system implemented the first-order version of the algorithm. Based on experience with this implementation, we concluded that Earley Deduction is not efficient enough to evaluate full ψ-logic programs. We did not implement a ψ-logic version of the algorithm and, instead, turned to development of the depth-first implementation described in Chapter 5. The second system implemented a version of Earley Deduction restricted to and optimized for the execution of Datalog programs.

In the next subsection, we discuss several indexing schemes employed in the first-order implementation. Then, we will discuss the implementation restricted to Datalog programs and describe several optimizations that can be applied to the Datalog case. The first-order implementation served as a useful control for evaluation of the Datalog

optimizations. Following the explanation of the optimizations, we present performance results. The Earley Deduction study was done using Smalltalk on a Tektronix 4400 personal workstation.

6.2.6.1. Indexing the Clauses

All program and derived clauses are indexed using three different keys to avoid searching all clauses during the subsumption check, the reduction step and the instantiation step. To speed up the subsumption check, a *complete key* is created. The complete key of a clause is the concatenation of the predicate names and arities of all literals occurring in that clause. For example, the clause:

```
p(X,Y) :- q(a,X,Y), r(f(X,Y)).
```

has the complete key **p-2-q-3-r-1**. To determine whether a clause is subsumed (see the definition of subsumption of clauses given earlier) by any existing derived clauses, only the clauses in the clause database with the same complete key need to be considered.

To facilitate reduction, we maintain second index of the non-unit program and derived clauses based on the *selected-literal key*, which consists of the predicate name of the selected literal and its arity. In this implementation, the selected literal is always the first (i.e., left-most) literal in the clause body. The selected-literal key for this clause is **q-3**. Each unit clause is used in an attempt to reduce all other clauses in the database. Since the unit clause must unify with the selected literal of other clauses, only those clauses with a selected-literal key equal to the complete key of the unit clause need to be retrieved.

Finally, for instantiation, an index for all non-unit program clauses based on the *program-rule-head key* is maintained. A program-rule-head key consists of the predicate name of the head (positive) literal and its arity. For the clause above, it is **p-2**. Given

a non-unit derived clause that we wish to use to instantiate program clauses, we compute a key based on its selected literal and arity. Then we need only consider those clauses with an identical program-rule-head index.

Example. To illustrate how the indices are organized, consider the deduction sequence shown in Figure 6.2. In Figure 6.9, the complete-key index is shown as it is after the deduction has terminated. In presenting an index, each key is followed by the clauses with that key. The actual clause is shown instead of a pointer to the internal representation of the clause. In the implementation, there is only one copy of each clause and the index entries point to those copies. Every clause is reachable from the complete-key index.

```
p-2-p-2-p-2
    p(X,Z)  :- p(X,Y), p(Y,Z).          (1)
    p(a,Z)  :- p(a,Y), p(Y,Z).          (6)
    p(b,Z)  :- p(b,Y), p(Y,Z).          (8)
    p(c,Z)  :- p(c,Y), p(Y,Z).          (11)

p-2-p-2
    p(a,Z)  :- p(b,Z).                  (7)
    p(b,Z)  :- p(c,Z).                  (10)
    p(a,Z)  :- p(c,Z).                  (13)

p-2
    p(a,b).                             (2)
    p(b,c).                             (3)
    p(a,c).                             (9)

ans-1-p-2
    ans(Z)  :- p(a,Z).                  (4)

ans-1
    ans(b).                             (5)
    ans(c).                             (12)
```

Figure 6.9: A Complete-Key Index

In Figure 6.10, the selected-literal key is shown. Note that not every clause appears in this index since only non-unit clauses have selected literals. Also note that, in this example, all non-unit clauses happen to have the same selected-literal key.

Finally, in Figure 6.11, the program-rule-head index is shown. There is only one program rule in this example deduction but, in general, there will be several distinct keys, each pointing to several clauses. □

6.2.6.2. Optimizations for Datalog Programs

In the second implementation, we explored additional optimizations to increase the algorithm's performance while restricting it to Datalog programs. In addition to the primary indices described above, secondary indices based on *format vectors* are maintained.

```
p-2
    p(X,Z)  :- p(X,Y), p(Y,Z).                               (1)
    ans(Z)  :- p(a,Z).                                       (4)
    p(a,Z)  :- p(a,Y), p(Y,Z).                               (6)
    p(a,Z)  :- p(b,Z).                                       (7)
    p(b,Z)  :- p(b,Y), p(Y,Z).                               (8)
    p(b,Z)  :- p(c,Z).                                       (10)
    p(c,Z)  :- p(c,Y), p(Y,Z).                               (11)
    p(a,Z)  :- p(c,Z).                                       (13)
```

Figure 6.10: A Selected-Literal-Key Index

```
p-2
    p(X,Z)  :- p(X,Y), p(Y,Z).                               (1)
```

Figure 6.11: A Program-Rule-Head Index

The format vector for a clause contains information about which argument positions are filled by constants and about variable usage in the clause. The format vector is a concatenation of items, one item for each argument position. The character "#" appears in the format vector in positions corresponding to arguments filled by constants. Variables are given normalized names (from 1, 2, ...) and these numbers appear in format vector positions corresponding to arguments filled by variables. For example, the format vector for the clause

```
p(a,X,Y) :- q(Y,b), r(X).
```

is #-1-2-2-#-1. Note that the predicate and arity information (which is contained in the primary keys) is not present in the format vector.

Given the complete key and format vector for a clause, all that is needed to fully specify the clause (up to renaming of variables) are the values of the constants, which are represented simply as a tuple.

Example. Figure 6.12 shows the representation of the clauses in the deduction of Figure 6.2. □

In a database with more than a few clauses that are equal up to constant values (and renaming of variables), this rather complex data representation saves space. Since many clauses with identical keys and format vectors are generated during a typical Datalog execution, this representation pays off. We will discuss performance in more depth below.

The main optimization for Datalog, however, is *compiling* the reduction, instantiation, and subsumption steps. When a new clause is generated, it becomes necessary to compare it with *all* existing clauses to see if it is subsumed by another existing clause. We call such a newly generated clause the *candidate clause*. If it is not sumbsumed, we

```
p-2-p-2-p-2
    #-1-#-2-2-1
        < a a >
        < b b >
        < c c >

p-2-p-2-p-2
    1-2-1-3-3-2
        < >

p-2-p-2
    #-1-#-1
        < a a >
        < b b >
        < c c >

p-2
    #-#
        < a b >
        < b c >
        < a c >

ans-1-p-2
    1-#-1
        < a >

ans-1
    #
        < b >
        < c >
```

Figure 6.12: The Representation of a Datalog Program

must then compare the candidate clause to *every* existing clause to see what new clauses can be derived from it using the reduction or instantiation rules. Given a candidate clause, we generate a sequence of "instructions" that can be "executed" to run through all existing clauses and perform the subsumption check quickly. Then, we generate a second sequence of "instructions" to run through all existing clauses and derive any new clauses using the reduction rule. Finally, we generate a third sequence of "instructions"

to run through the existing clauses and derive any new clauses by instantiation.

Consider the subsumption check first. We must look through the primary indices and, for each, we must look through all format vectors. Associated with each format vector is a set of tuples S. Each tuple represents one clause. We perform one compilation for each set S and then execute the "instructions" once for each tuple. Since all the tuples (clauses) in S have the same key and same format vector, the subsumption check can be done for all the tuples in S at once by abstracting away from the actual values of the constants. The result of such the compilation is a sequence of equality checks, which can then be evaluated quickly for each of the tuples in S.

To compile the subsumption check, we use an algorithm that, given two clauses, determines whether the subsumption relation holds. One of these clauses — the candidate clause — is known fully but the other clause will not be known fully until "run time". Instead, the compilation uses a symbolic clause to represent any clause in S. We do not know the exact values of the constants, but we know everything else about the clause. Thus, when the value of one of these constants must be examined in the subsumption check, we *defer* the check until run time.

Example. Let the candidate clause be

```
p(a, X, Y) :- q(Y).
```

and let all the clauses in S have the form

```
p(#, #, X) :- r(Y, #).
```

For clarity, we use a symbolic clause containing the symbol #. In fact, the compilation step is given only the complete key **p-3-r-2** and the format vector **#-#-1-2-#**. Regardless of the values of the constants, the candidate clause clearly will not be subsumed by any clause in S. We don't need to examine the individual tuples. □

Example. Let the candidate clause be

```
p(a, b, c) :- q(b).
```

and let all the clauses in S have the form

```
p(#1, Z, #2) :- q(Z).
```

We have sequentially numbered the constants in S to identify their positions in the tuples < $\#_1$ $\#_2$ >. We first notice that $\#_1$ must be a in order for the candidate clause to be subsumed. This equality check is deferred. Likewise, the check $\#_2$ = c is deferred. Finally, we note that the substitution { Z ← b } is consistent with the subsumption order. The result of the compilation is the set of instructions

$$\#_1 = a$$
$$\#_2 = c$$

That is, if S contains the tuple

```
< a c >
```

then the candidate clause is subsumed by an existing clause and need not be added. After compiling, we "execute" these 2 instructions once for every tuple in S. Testing these two equalities is much faster than performing a full subsumption check. □

In the case of a reduction step, we will use the candidate clause (a unit clause) to reduce all of the non-unit clauses in S. To perform all these reductions at once, we must perform the reduction of a symbolic clause, i.e., a clause in which the actual constant values are unknown and are represented by $\#_i$ at compile time.

There are two cases where these values are needed: when we compare the value to another value and when we use the value in a substitution. As before, we will defer value checking until run time, when we get the values from a tuple in S. For substitutions, we don't need the actual value. We can just use the symbolic $\#_i$ value.

When we construct the clause resulting from the reduction step, we end up with a clause that may contain both actual values (from the candidate clause) and symbolic values (from the symbolic clause). In any case, we can determine the complete key and the format vector of the result. Then, given the actual values of the constants at run time, we can easily build a tuple to represent the result clause.

Example. Consider the candidate clause:

```
q(a, b, b, U, U, V, V).
```

This clause must be used to reduce all clauses with a seven-placed predicate named `q` as the selected literal. To find these clauses, we first use the selected-literal index, which will retrieve all those clauses with complete keys of the form *x*-*x*-**q-7**-*x*-*x*-.... One such key is **p-3-q-7-r-3** and it will be used for this example.

Associated with this key are several format vectors. We will look at

1-2-#-#-2-2-#-#-3-4-4-#-2

For example, the clauses

```
p(W,X,a)  :- q(b,X,X,c,d,Y,Z), r(Z,e,X).
p(U,V,a)  :- q(a,V,V,c,c,Y,W), r(W,e,V).
p(W,U,a)  :- q(b,U,U,d,d,Y,Z), r(Z,e,U).
p(Z,Y,c)  :- q(a,Y,Y,b,b,W,X), r(X,d,Y).
```

would be represented by the tuples

```
< a b c d e >
< a a c c e >
< a b d d e >
< c a b b d >
```

These tuples comprise S. Call the tuples (clauses) in S the *target* tuples (clauses).

The compilation phase may fail, in which case we know that none of the target tuples unify with the candidate tuple, without ever looking at any of the target tuples.

In this example however, the compilation succeeds producing the following "instruction" sequence:

$$\#_2 = a$$
$$\#_3 = \#_4$$

These instructions say that any tuple with the constant a in the second position and with the third and fourth positions equal represents a clause in which the selected literal unifies with the candidate clause.

For every such tuple we must construct a new derived clause. By removing the selected literal from the target clauses, we see that the derived clause will have the key **p-3-r-3**. The compilation phase also produces a format vector describing the new clauses (**1-#-#-2-#-#**) along with the following information telling how to construct the new derived tuples from the target tuples:

$$\#_1 \leftarrow b$$
$$\#_2 \leftarrow \#_1$$
$$\#_3 \leftarrow \#_5$$
$$\#_4 \leftarrow b$$

Each tuple has 4 components and these "instructions" tell what values to use. On the right-hand sides, $\#_1$ and $\#_5$ refer to constant values from the target tuples.

After the compilation is complete, the equality comparisons are evaluated for each of the target tuples. The second and fourth tuples satisfy them. The following derived tuples can then be constructed using the tuple creation information:

```
< b a e b >
< b c d b >
```

These tuples represent the desired clauses:

```
p(X,b,a) :- r(Y,e,b).
p(X,b,c) :- r(Y,d,b).
```

□

The procedure to compile a number of instantiation steps is almost identical to the compiling of reduction steps. Given a candidate clause and a target set S representing a number of clauses identical up to constant values, the unification and substitutions are performed using the symbolic constant values, $\#_i$. The result is again a number of equality checks and a sequence of instructions showing how to produce a tuple representing a derived clause, given a target tuple.

Example. Given the candidate clause

```
P(X, Y) :- q(X, a, X, b), r(c, Y).
```

and a target set S of clauses of the form

$$q(\#_1,\ \#_2,\ \#_3,\ X)\ \text{:-}\ s(X,\ \#_4),\ t(X,\ Y).$$

the following equality checks result

$$\#_2 = a$$
$$\#_1 = \#_3$$

The derived clauses are of the form

$$q(\#_1,\ \#_2,\ \#_3,\ \#_4)\ \text{:-}\ s(\#_5,\ \#_6),\ t(\#_7,\ X).$$

The tuples representing the derived clauses are constructed from the target tuples as follows:

$$\#_1 \leftarrow \#_1$$
$$\#_2 \leftarrow \#_2$$
$$\#_3 \leftarrow \#_3$$
$$\#_4 \leftarrow b$$
$$\#_5 \leftarrow \#_4$$
$$\#_6 \leftarrow b$$
$$\#_7 \leftarrow b$$

□

These operations — comparing and manipulating tuple values — are available in relational algebra [Maier 1983]. Although our implementation did not use relational operators, and creating new tuples representing reduced clauses for any tuples found to satisfy the comparisons can always be expressed using standard relational operators.

Example. Consider the example above illustrating the the compilation of the reduction step. The compiled instructions included the equality checks

$$\#_2 = a$$
$$\#_3 = \#_4$$

and the instructions for building the result tuples

$$\#_1 \leftarrow b$$
$$\#_2 \leftarrow \#_1$$
$$\#_3 \leftarrow \#_5$$
$$\#_4 \leftarrow b$$

If we label the positions of the target tuples with the attribute names $A_1, A_2, \cdots A_5$ and call the set of target tuples the relation s, the set of tuples representing the derived clauses can be represented (using the notation of [Maier 1983]) as:

$$\delta_{A_2 A_3 \leftarrow A_1, A_5}(\pi_{\{A_1, A_5\}}(\sigma_{A_2=a}(s[A_3=A_4]s))) \bowtie <b{:}A_1\ b{:}A_4>$$

□

Earley Deduction will terminate for Datalog programs even if the subsumption check is relaxed to an equality check. In that case, a new tuple is not added to the derived set if it is already there. By using a hash table index for the individual tuples, this check can be done in essentially constant time. We will save time when the time saved by using the equality check outweighs the additional time associated with processing extra clauses that would have been deleted by the full subsumption check. We implemented this optimization to see whether it saved time and our results are given in

the next section.

Another optimization we implemented involves batching up the subsumption checking. In the course of a reduction (or instantiation) step, a number of clauses with identical keys and format vectors will be created. The subsumption check must be performed on each of these before it can be added to the derived set. By delaying the subsumption checking until one of these tuples is referenced, a number of very similar compilations can be replaced with a single compilation. Then the subsumption check for all of the clauses is performed at one time by repeatedly executing the compiled instructions.

6.2.6.3. Experimental Evaluation of the Optimizations

All of the optimizations described above were implemented and a several experiments were performed to determine whether and how much each optimization speeded up the Earley Deduction algorithm.

In the first experiment, the program in Figure 6.13 was used. The figure shows only one p unit clause symbolically; there were actually several p clauses. We varied the number of p clauses from 3 to 40, choosing the constants a_i and a_j randomly from the set

```
s(X, X).
s(X, Y) :- p(X, U), s(U, V), p(Y, V).
p(ai, aj).
:- s(X, Y).
```

Figure 6.13: A Test Program

{ a, b, c, d, e, f, g }

For each such program, we plotted the time required for the first-order version of the algorithm to terminate. (Since all our programs were Datalog programs, all deductions terminated.) For a given number of p clauses, we actually generated several programs, and ran each program to give a range of running times. These data are given by the upper curve (labeled *Without Datalog Optimizations*) in Figure 6.14. The vertical thickness of the upper curve reflects the range of running times observed.

The lower curve (labeled *With Datalog Optimizations*) shows the improvement realized in the second implementation of the algorithm, optimized for Datalog programs as described above. The same programs used to generate the upper curve were used to generate the lower curve.

The second experiment used the program in Figure 6.15. Again, we varied the number of unit clauses p and, for each number, generated several programs by randomly selecting constant values from a set of 7 constants. The times to complete the deduction by both the first-order version of the algorithm and by the Datalog version of the algorithm are plotted in Figure 6.16.

In the third experiment, we asked whether substituting the equality check for the full subsumption check in the Datalog version improved the running times. We used the program given in Figure 6.15, again randomly choosing the a_i and a_j from a set of 7 constants and varying the number of unit clauses. We also asked whether batching up the subsumption checking (i.e., doing several subsumption checks at once, rather than checking each newly derived clause at a time) improved execution time. The results are shown in Figure 6.17. Although the baseline curve (labeled *Basic Datalog Optimization*) is quite lumpy — due to wide variation in the difficulty of the sample programs — the

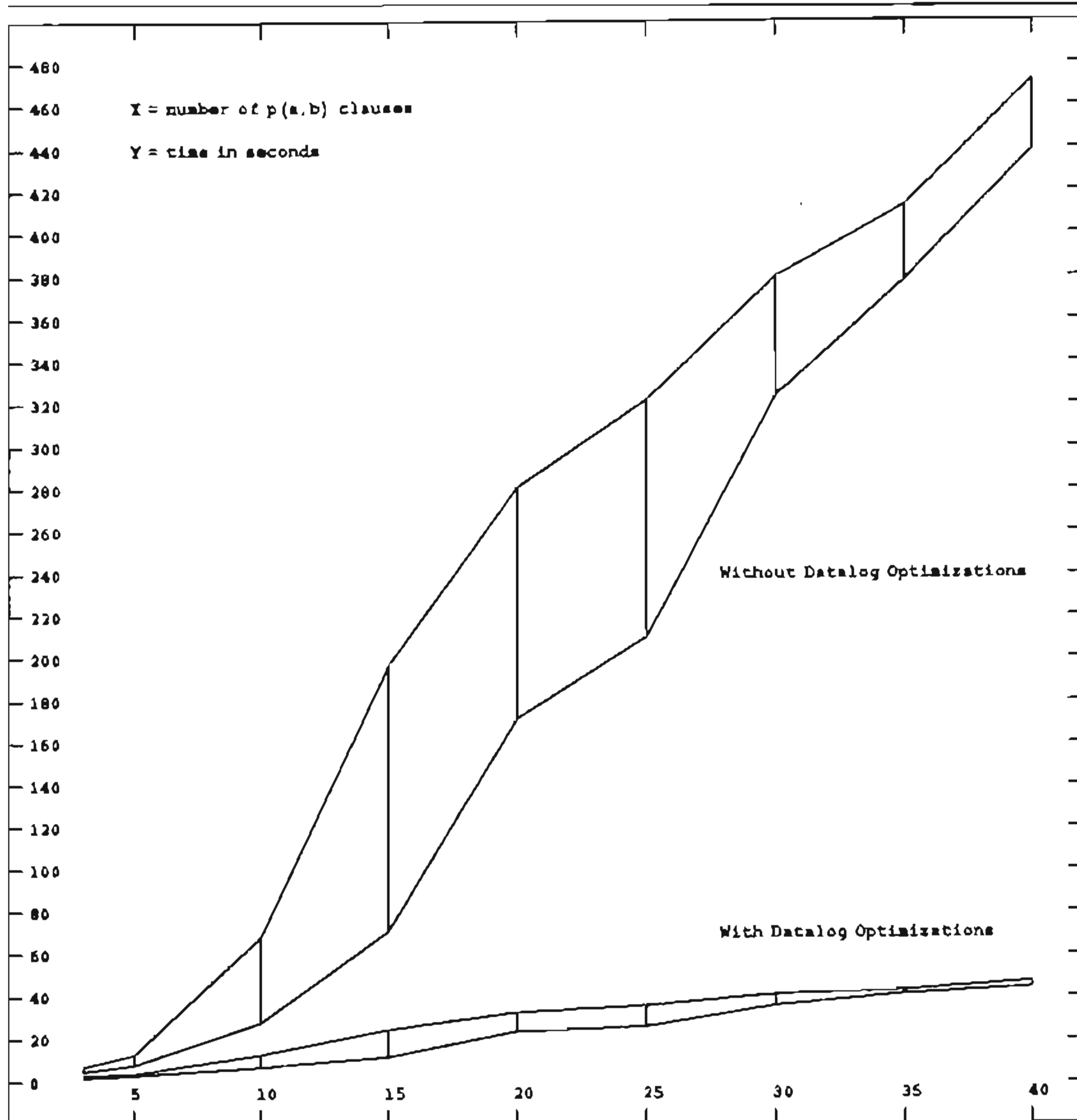

Figure 6.14: The Execution Times for Figure 6.13

```
s(X, Y) :- p(X, Y).
s(X, Y) :- p(X, Z), s(Z, Y).
p(a_i, a_j).
:- s(b, Y).
```

Figure 6.15: Another Test Program

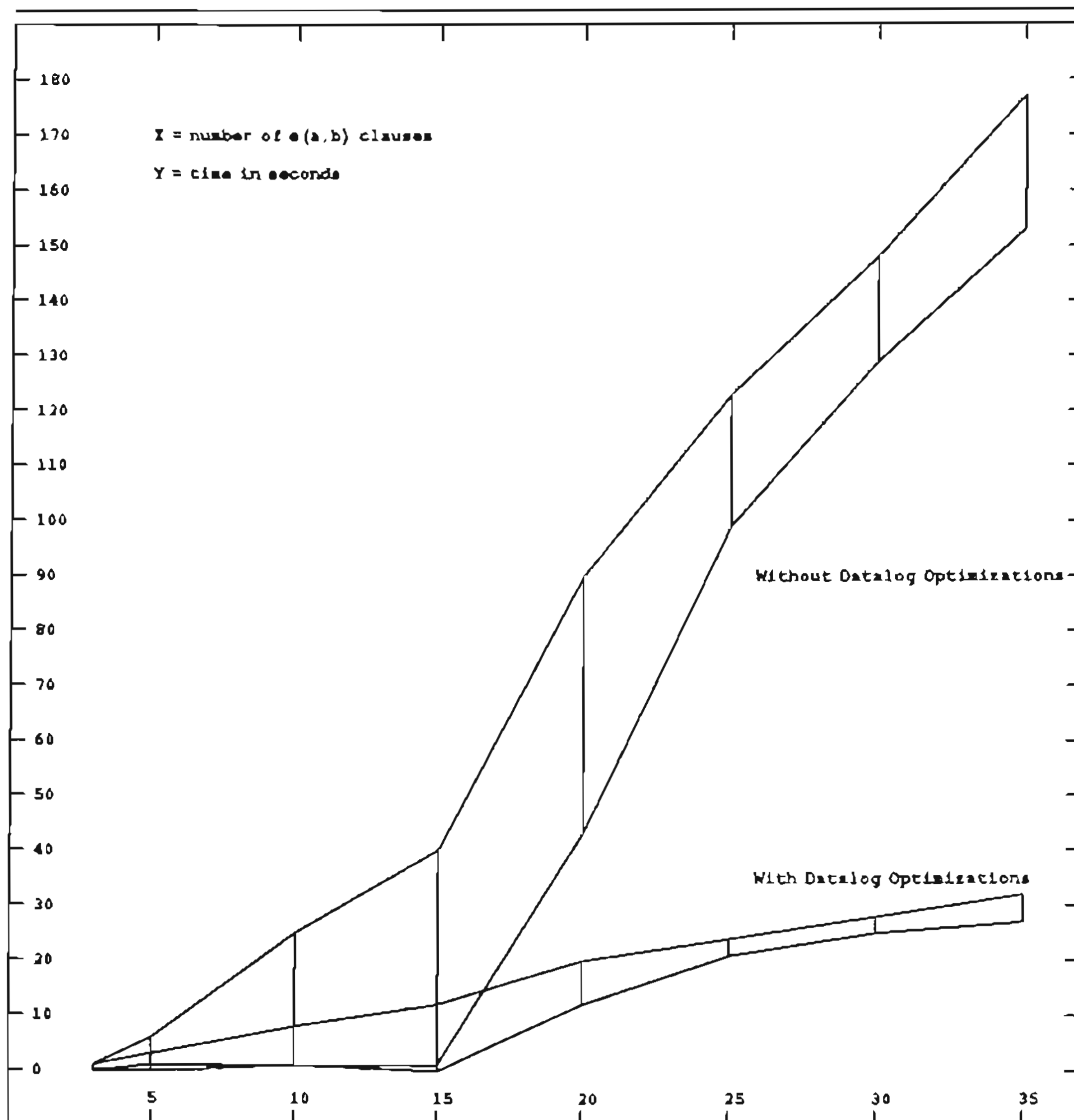

Figure 6.16: The Execution Times for Figure 6.15

offsets of the other two curves is fairly small and consistent.

Based on these three experiments, we can draw several conclusions about the Earley Deduction algorithm. First, for first-order programs that can be executed using a

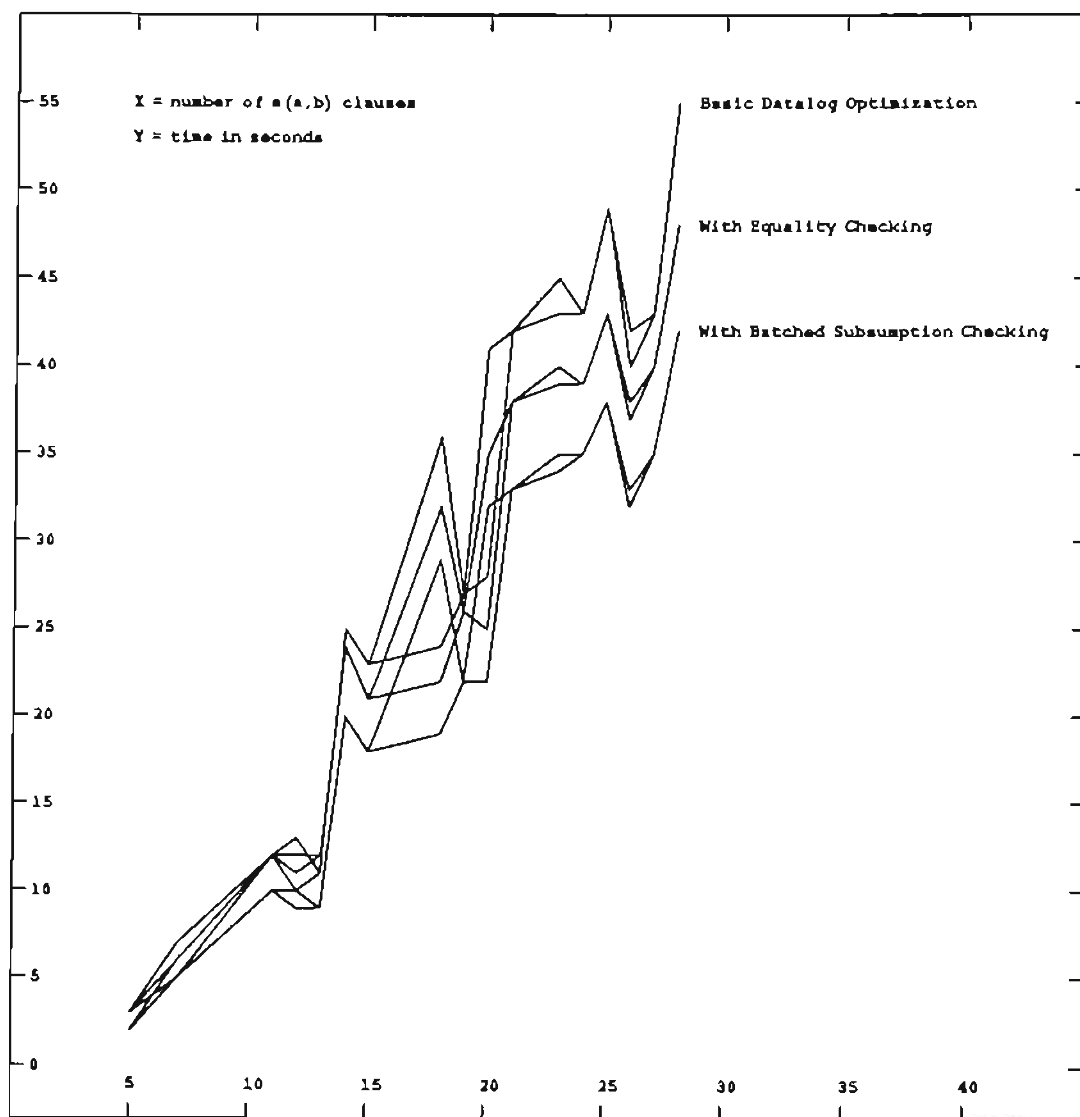

Figure 6.17: Analysis of Subsumption Checking

top-down strategy[6], Earley Deduction is not nearly fast enough to compete with typical Prolog interpreters, nor do we believe it can be made fast enough. Its usefulness is in executing logic programs without concern for the order of the program clauses (or

[6]That is, when a top-down strategy will produce all desired answers and will not cause non-termination.

literals within the clauses) or when all solutions are wanted in the face of possible non-termination.

Second, the basic Datalog optimization (representing the clauses as tuples and compiling the reduction step, the instantiation step and the subsumption check), resulted in a significant speed-up over the first-order version of the algorithm — over 10 times for some of the small programs tried. Furthermore, the improvement realized from the Datalog optimization increases as the length of the deduction grows, since compiling has a greater benefit the more times the compiled instructions are executed.

Finally, replacing the subsumption check with the simpler equality check only speeded up the algorithm a little. Replacing the equality check by the batched subsumption check improved performance a little more. Since these two optimizations resulted in only small increases, it is unclear whether a significant improvement would be obtained for other programs. In some cases, the difference in running time between the different optimizations was less than the difference in running time between different programs within one version of the algorithm.

Our experiments were performed entirely within the Smalltalk environment. Using the representation of Datalog clauses described above, it would be fairly straightforward to store the tuples in a traditional relational database. The compilation and overall system organization would comprise a front-end and would access the tuples using only the standard relational operators. In this way, the system could be enhanced to handle very large Datalog programs. The implementation and empirical evaluation of such a hybrid system (vis-a-vis a Prolog interpreter) is one direction for further research.

6.3. Extension Tables

In this section, we describe an evaluation strategy S.W. Dietrich and D.S. Warren introduced for executing first-order logic programs and discuss its application to ψ-logic [Dietrich and Warren 1986]. The method, which can be viewed as a generalization of the use of *memo functions*, uses *extension tables* to save results and, thus, avoid the recomputation of answers. In summary, the idea applies the principles dynamic programming to the execution of logic programs.

Let us first review how memo tables are used in the context of functional programming by considering the evaluation of some function f. In addition to the code to compute the function, a table is also associated with f. This table is initially empty. The first time f is called with some argument x, the function is evaluated as usual to produce a result y. Before returning y to the caller, a single entry is added to the table associated with f. Each table entry is a pair of values. An entry is added for this invocation of f containing both the argument x and the computed result y. The table is indexed on the argument field.

Then, on a subsequent call to f with argument x', we first check the table. If the table contains an entry for x', we need not compute the corresponding result. Instead, we just return the result stored in the table.

How do memo tables affect the running time of f? Consider, for example, the following functional definition of Fibonacci numbers:

```
fib(X) =
    if X=1 then
        1
    elseif X=2 then
        1
    else
        fib(X-1) + fib(X-2)
    endif
```

Without the memo table, `fib` requires exponential time to execute, because each level of recursion multiplies the number of invocations by two. With a memo table, the first subroutine call within `fib` (to `fib(X-1)`) will compute all values of `fib(1)` ... `fib(X-2)` and save them in the table. The second call (to `fib(X-2)` can then be satisfied by a table lookup. Thus, with the table, `fib` executes in approximately linear time, assuming a hash table index for the table.

When the concept of memo tables is applied to logic clauses instead of functions, the tables are called *extension tables* since they store extensions of predicates. One extension table is maintained for each predicate. The several clauses comprising the predicate's definition are all associated with this one table.

Let's consider what happens when we maintain an extension table for some predicate **p**. Let $\overline{X}$ represents the vector of arguments X, Y, ... , Z to predicate **p**. When **p** is called with arguments $\overline{X}$, we will check the table associated with **p** for an entry indexed by $\overline{X}$. Assume that no matching entry exists yet. We will then compute the answers for $\mathbf{p}(\overline{X})$ and store them in a newly created table entry. These answers are instantiated versions of the arguments $\overline{X}$. In general, there will be many answers for any one argument $\overline{X}$ vector, which we will denote $\overline{X}_1$, $\overline{X}_2$, ... The table entry will then contain both the original arguments, $\overline{X}$, and all the solutions $\overline{X}_1$, $\overline{X}_2$, ... For simplicity, assume that there are a finite number of solutions, so they can be computed finitely and stored in a finite table entry. (Don't worry, we will deal with this assumption later.)

To reduce the size of the extension tables and to reduce the number of redundant answer solutions returned, when adding a new solution the algorithm first checks the solutions already in the table and elides duplicate solutions.

Example. Consider the following program:

```
p(X, Y, Z) :- q(Z, Y, X).
q(c, d, a).
q(e, f, a).
q(c, g, a).
q(b, a, b).
```

Evaluation of the query

```
:- p(a, U, V).
```

results in the following extension table entry for p.

```
p(a, U, V):
     [ p(a, d, c), p(a, f, e), p(a, g, c) ]
```

□

After exhaustively computing the extension of $p(\overline{X})$, each solution $\overline{X}_i$ can be returned in turn to the invoking process. On subsequent calls to p (say with arguments $\overline{X}'$), we first check the extension table for p for an entry indexed by $\overline{X}'$. If one exists, we need not recompute the solutions; we can just return the solutions stored in the table. We call this *borrowing* an existing entry.

Actually, when we check the extension table to see if we have already called p with the current arguments $\vec{X}'$, we don't need an exact match. If we have previously created an entry for arguments $\vec{X}$ and if $\vec{X}$ is more general than $\vec{X}'$, then we have already solved a more general problem, of which the current arguments represent a special case. We do not need to redo the work: we can make use of the solutions already computed. (By $\vec{X}$ *more general than* $\vec{X}'$, we mean that every argument of $\vec{X}$ is more

general than the corresponding element of $\overline{X}'$.)

But we cannot return all the solutions for $p(\overline{X})$. Some of them may not even be subsumed by $\overline{X}'$ and a solution is always subsumed by the arguments. We can return any solution $\overline{X}_i$ in the table entry for $\overline{X}$ that is subsumed by $\overline{X}'$.

Example. In our example, a call to evaluate

```
:- p(a, U, c).
```

can use (borrow) the table entry above. The answers

```
p(a, d, c)
p(a, g, c)
```

can be returned as solutions, but

```
p(a, f, e)
```

cannot be. since it is not subsumed by `p(a, U, c)`. □

The extension table algorithm as described so far has a couple of problems. First, we assumed that there were a finite number of solutions to goal $p(\overline{X})$: there may be an infinite number. Even if there are only finitely many, we may be interested in just the first one or two solutions.

To avoid computing more than necessary (and perhaps failing to terminate and thus missing other answers that could be produced), the algorithm just computes the first (or next) solution. This solution is then added to the table and returned. The table entry is marked as *incomplete* to indicate that there may be additional solutions. If the first solution turns out to be inadequate and backtracking forces us to find another answer, the incomplete computation can be resumed to produce another solution. The table entry is extended with this new solution and it is returned. If, instead, no more solutions can be found, the table entry is marked *complete* and the call to p

fails.

Second, we tacitly assumed that there was no interaction between calls. When p was called, we assumed the algorithm backtracked it through all solutions until the table entry was complete. Only then would other calls to p occur. In practice, we may encounter a call to $p(\overline{X}')$ before we have completed building the table entry for $p(\overline{X})$, where $\overline{X}'$ is an instance of $\overline{X}$. If p is defined recursively, the second invocation may occur while we are trying to compute a new solution for p to be added to the table. Whether the call is recursive or not, we can use any existing solutions in the table entry before trying to compute another entry directly. The prescription for the call to $p(\overline{X}')$ is to use all the solutions currently in the table entry created by the call to $p(\overline{X})$ that are instances of $\overline{X}$. If, on backtracking, more solutions are required and the table entry is marked *incomplete*, we must then compute the answers directly.

There are two variations of the algorithm for handling this situation: we can compute another solution to the query $p(\overline{X}')$ or we can compute another solution to $p(\overline{X})$. Since it is $p(\overline{X}')$ that we are interested in, it makes sense to compute solutions for it. Attempting to compute another answer for the more general $p(\overline{X})$ could get us into non-termination, while computing another solution for $p(\overline{X}')$ might not.

Example. Consider the following program:

```
r(U, V) :- p(X, Y, Z), q(Y), p(Z, U, V).
p(c, b, d).
p(a, d, c).
p(g, X, Y) :- p(g, Y, X).
p(c, e, f).
q(d).
:- r(U, V).
```

For the first call to `p` in the body of the `r` rule, all three arguments are uninstantiated. We create a table entry indexed by `p(X, Y, Z)`, find the first solution `p(c, b, d)`,

add it to the entry, and return it. Because `q` fails, we backtrack, find the second solution `p(a, d, c)`, and add it to the table entry. This time `q` succeeds and we call a new goal `p(c, U, V)`. We can use the table entry indexed by `p(X, Y, Z)` to print the first solution `r(b, d)`. On backtracking, the second solution `p(a, d, c)` is not an instance of the goal `p(c, U, V)` so it is skipped. At this point we have exhausted the table entry for `p(X, Y, Z)` but need more solutions. Attempting to find another solution for `p(X, Y, Z)` would then use the recursive rule and get into an infinite loop. Attempting to find another solution for `p(c, U, V)` results in finding another answer `r(e, f)`. □

However, since we have done no actual computation for $p(\overline{X}')$ (in the example, `p(c, U, V)`), there is no backtrack point to return to! The procedure must walk through solutions already obtained (such as `p(c, b, d)`) until a novel solution is found. This new solution cannot be added to the table entry, since the table entry is for $p(\overline{X})$ not $p(\overline{X}')$. Since we have to compute all the previously returned solutions for $p(\overline{X}')$ anyway, a new table entry indexed by $p(\overline{X}')$ can be created. The recomputed solutions are added to this entry and the first novel solution is returned. The other option — to compute more entries for $p(\overline{X})$ — is problematic since the current context is $p(\overline{X}')$ not $p(\overline{X})$. To take this approach, each incomplete table entry is augmented with an additional field. After each solution of $p(\overline{X})$ is found, the extension table procedure stores a *continuation* in the table entry, which can be *resumed* to produce additional solutions as necessary. The additional work to compute $p(\overline{X})$ rather than $p(\overline{X}')$ must be done eventually if we are searching for all answers anyway (and the program is *pure* in the sense that it does not contain cuts, etc.). Consequently, the second option, while more difficult to implement, is preferred.

We conclude the description of the extension table algorithm by looking at one special case for which we can detect infinite looping. Consider what happens when we are computing solutions for $p(\overline{X})$. Assume we have exhausted any solutions stored in the table and have turned to the continuation to find the next solution. Perhaps, in the course of finding this solution, we call $p(\overline{X})$ itself and again we turn to the table entry just mentioned. After exhausting all solutions, we are left with the continuation to compute the next entry. But, it is that continuation that we are currently executing! An attempt to use it again will produce an infinite loop. By adding a flag to each table entry, this case of non-termination can be detected and avoided.

Example. Consider the following program:

```
p(a, b).
p(X, Y) :- p(Y, X).
:- p(U, V).
```

We first create a table entry for the goal `p(U, V)`, find a solution `p(a, b)`, and store it in the table entry along with a continuation. When the user requests backtracking, we resume the continuation which calls a new goal `p(Y, X)`. We begin by borrowing the previously created table entry and find the solution `p(a, b)`. At this point, we suspend the goal `p(Y, X)`. In the appropriate activation record, we store an indication that, upon backtracking, we should return the second solution in this table entry (if there is one) as the next solution to `p(Y, X)`. We then return this answer to the calling goal `p(X, Y)`. This leads to a new solution for `p(X, Y)` – namely `p(b, a)` – which we store as the second solution in the table entry for `p(X, Y)` and return.

When the user invokes backtracking to find a third answer, we again resume the continuation to find another solution to `p(X, Y)`. We immediately backtrack to the goal `p(Y, X)`. Since a new solution `p(b, a)` has appeared in the table entry, the

goal `p(Y, X)` returns it. But, since this leads to no new solutions for `p(X, Y)`, we again backtrack to the goal `p(Y, X)`. Remember that `p(Y, X)` was using a borrowed table entry which it has now exhausted. So `p(Y, X)` looks to the continuation for another answer. But this continuation is currently being executed.

If we were to copy the continuation and restart the second copy, it would only do what the first copy did, namely reach this continuation and try to restart it. Instead, the extension table algorithm can either report back to the user that looping was detected in the use of a recursive rule or backtrack even further in search of another solution. □

Applying the extension table algorithm as described to ψ-logic is straightforward, since the only operations being performed are table lookup and modification and comparing goal literals to each other to check for subsumption. The indexing of the extension tables is the only thing that is not completely straightforward. For example, a table entry for the goal ψ-literal

```
p( ... )
```

might be useful in providing answers for an apparently unrelated goal ψ-literal

```
q( ... )
```

But this problem exists in any implementation when a clause must be retrieved from the database to solved a given goal ψ-literal. In our depth-first implementation, we solved it by using *maybe sets* (see Chapter 5). Maybe sets could also be used in indexing extension table entries.

Dietrich and Warren claim that their extension table algorithm is complete for Datalog programs. We conjecture that, when an extension table is maintained for every predicate, the procedure as described above is complete for the more general cases of

first-order logic and ψ-logic. However, the algorithm is complex and completeness is neither obvious nor proven. Dietrich and Warren discuss a variation, called ET*, that combines aspects of extension tables with the staged depth-first search strategy (to be discussed in the next subsection). The completeness of ET* follows from the completeness of the staged depth-first search strategy.

6.4. Staged Depth-First Search Strategy

In [Stickel 1984], Mark Stickel describes a complete evaluation strategy he calls *staged depth-first search strategy* that is used in his Prolog Technology Theorem Prover. The idea is much simpler than Earley Deduction and the extension table algorithm. First, a depth bound d is maintained during the search for a proof. The proof tree is not allowed to exceed this depth. (A tree's depth is the length of the longest branch from root to leaf.) If the depth of a proof tree exceeds d, that branch of the search tree is pruned and other, shallower proofs are sought.

First, the depth bound d is initialized to some constant. Then, all proofs of depth less than d are sought. This process terminates after any and all such proofs are found. Then, the depth bound is increased (say to $d+1$) and the process is repeated. Clearly, every proof will ultimately be found, since d will ultimately exceed the depth of every proof tree. And obviously, when one pass of the algorithm searching for proofs of depth less than some value of d, terminates without being cut off by the depth-bound, it is clear that the entire procedure can be terminated. Since the depth bound was not reached, there are no more proofs to be found.

The staged depth-first search strategy can be adapted to the evaluation of ψ-logic programs very simply. Given a depth-first execution environment for ψ-logic, such as the systems described in Chapter 5, all that needs to be done is to maintain the depth

of the current proof. This depth can be stored in an additional field of the activation record. When each activation record is allocated, its depth is set to 1 greater than the depth of the calling activation record (see Chapter 5), and is checked against the current depth bound.

The drawback of the staged depth-first search strategy is that the search with the depth bound set to d repeats all the work of the previous pass. Stickel shows that, with a sufficient branching factor in the search space, the work associated with depth $d+1$ will greatly exceed the work for depth d. Unfortunately, some proofs may include substantial amounts of linear reasoning in which the branching factor is 1.

6.5. Other Evaluation Strategies

There is other work on evaluation strategies for first-order logic program execution that may be adapted to the execution of ψ-logic programs. Although we have not explored these connections, we mention several of these efforts next.

Jeffrey Ullman and students have developed the NAIL! system for the execution of functor-free programs [Ullman 1984, 1985]. Summarizing, the system compiles queries by building a *rule/goal graph* describing the clauses and their interconnectivity. The system then analyses this graph trying to find a relational expression that effectively computes the answer relation. The desired relational expression can be produced if nodes in the tree can be *captured* and a number of *capture rules* are described. L. J. Henschen and S. A. Naqvi also describe an algorithm for compiling queries in the functor-free subset of logic [Henschen and Naqvi 1984].

D.E. Smith, M.R. Genesereth and M.L. Ginsberg define a recursive inference as an infinite sequence of goals containing repeated similar subgoals. By examing goal sequences, certain portions of the search space can be identified as unable to produce

any new answers. They present a number of theoretical results about where the search for a proof may safely be pruned [Smith, et al. 1986]. In addition, they also present a number of provocative examples while discussing infinite inference chains and techniques for dealing with them.

6.6. Discussion of Evaluation Strategies

We have described Earley Deduction (including results on soundness, completeness, termination for Datalog, and implementation), the extension table algorithm, and the staged depth-first search strategy and have mentioned other evaluation strategies. Research in evaluation strategies, particularly for the limited Datalog case, is an active area. Many of the execution strategies developed for first-order logic can be extended directly to Inheritance Grammars, since they make no assumptions about the underlying logic that are not satisfied by ψ-logic. Which strategies are most appropriate for executing Inheritance Grammars?

In the Chapter 5, we discussed our implementation of ψ-logic which is based on the depth-first search strategy. This strategy is fast but incomplete and so the implementation is an incomplete one. We have not attempted to evaluate complete strategies; we can only speculate and leave the question open.

Earley Deduction is, in some sense, the purest of the strategies. Because of its simplicity, it may be most amenable to hardware/parallel implementations and may hold the most potential. As it stands, however, it is too costly to be of more than academic interest.

The staged depth-first search strategy is probably the easiest to implement but, because it repeats work, will probably be outperformed by Earley Deduction and the extension table algorithm. For grammatical parsing applications, the extension table

algorithm may be the most effective of the complete strategies for two reasons.

First, the extension table algorithm has some flexibility the others do not. For many trivial predicates, it is much faster to recompute results rather than to save answers and try to reuse them, especially when the predicates have very large extensions, when the search trees are very shallow, and when a fast depth-first search strategy can be used. Only for a few predicates will it pay to save partial results. The extension table algorithm can accommodate this by only building the tables for *nasty* predicates. The grammar writer might, for example, flag certain problem predicates for extension tables in the process of tuning a grammar for faster execution. (It would be nice if nasty predicates could be identified automatically, but it is unclear how this could be done. Perhaps one could devise a nastiness metric based on number of clauses, number of literals, recursiveness, and the nastiness of called clauses.) Second, it may be easier for the grammar writer to form a mental model of how the algorithm works, thus enabling him debug or modify an existing grammar more effectively.

Conclusions and Future Directions

This dissertation introduced a new formalism, *Inheritance Grammar*, in which to express grammars for Natural Languages. We provided a formal semantics based on ψ-logic, described our implementations, discussed other implementation techniques and, in Appendix 4, presented an extended English Grammar using the Inheritance Grammar formalism. What next?

First and foremost, logic-based and unification-based formalisms need to address the issue of *negative* and *disjunctive* information in greater depth. For example, we would like to write a term

```
p (f⇒~q)
```

to mean "those p's that do not have q as a value of feature f." And we would like to write

```
p (f⇒{q,r,s})
```

to mean "those p's that have q, r, or s as a value of feature f." These changes to the formalism open up a whole new can of worms in the semantics, which we have not addressed. While additional work could surely be done to polish the formal semantics of ψ-logic as it now stands, we feel that revising the definition of ψ-logic to encompass negation and disjunction would be a more fruitful course.

A second research approach is to do some "field testing" of Inheritance Grammar by incorporating it into a production-scale Natural Language understanding system. Our execution environment is stable and considered complete but, under heavy usage,

the implementation of the ψ-interpreter might become an important issue. For example, we do not know whether it would be more profitable to invest additional work on our system in increasing execution speed or in adding new functionality (e.g., adding more built-in predicates).

Well... anyway...

The
End

References

Abramson 1984a

Abramson, Harvey, Definite Clause Translation Grammars, *Symposium on Logic Programming*, Atlantic City, NJ, 1984.

Abramson 1984b

Abramson, Harvey, Definite Clause Translation Grammars and the Logical Specification of Data Types as Unambiguous Context Free Grammars, Tech. Report 84-11, Dept. of C.S., Univ. of British Columbia, Vancouver, BC, 1984.

Aho and Ullman 1972

Aho, Alfred V., Ullman, Jeffrey D., *The Theory of Parsing, Translation, and Compiling*, vols. 1 and 2, Prentice Hall, Englewood Cliffs, NJ, 1972.

Aho and Ullman 1977

Aho, Alfred V., Ullman, Jeffrey D., *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1977.

Ait-Kaci 1984

Ait-Kaci, Hassan, A Lattice Theoretic Approach to Computation Based on a Calculus of Partially Ordered Type Structures, Ph.D. Dissertation, University of Pennsylvannia, Philadelphia, PA, 1984.

Ait-Kaci and Nasr 1986

Ait-Kaci, Hassan and Nasr, Roger, LOGIN: A Logic Programming Language with Built-in Inheritance, *Journal of Logic Programming*, 3(3):185-216 (1986).

Apt and Van Emden 1982

Apt, Krzysztof R., Van Emden, M.H., Contributions to the Theory of Logic

Programming, *J. ACM*, 29(3):841-862 (1982).

Bates 1978

Bates, Madeline, The Theory and Practice of Augmented Transition Network Grammars, in *Natural Language Communication with Computers*, Leonard Bolc (ed.), Springer-Verlag, New York, 1978.

Birkhoff 1979

Birkhoff, Garrett, *Lattice Theory*, American Mathematical Society, Providence, RI, Third Edition, 1979.

Bresnan and Kaplan 1982

Bresnan, Joan, and Kaplan, Ronald M., Lexical-Functional Grammar: A Formal System for Grammatical Representation, in *The Mental Representation of Grammatical Relations*, MIT Press, Cambridge, MA, 1982.

Colmerauer 1978

Colmerauer, Alain, Metamorphosis Grammars, in *Natural Language Communication with Computers*, Leonard Bolc (ed.), Springer-Verlag, New York, 1978.

Colmerauer 1982

Colmerauer, Alain, An Interesting Subset of Natural Language, in *Logic Programming*, K.L.Clark and S.-A.Tarnlund (eds.), Academic Press, London, 1982.

Colmerauer 1986

Colmerauer, Alain, Theoretical Model of Prolog II, in *Logic Programming and Its Applications*, Michel van Caneghem and David H.D. Warren (eds.), Ablex Publishing Co., Norwood, NJ, 1986.

Codd 1970

Codd, E.F., A Relational Model of Data for Large Shared Data Banks, *Comm. ACM*, 13(6):377-387 (1970).

Dahl 1979

Dahl, Veronica, Logical Design of Deductive NL Consultable Data Bases, *Proc. 5th Intl. Conf. on Very Large Data Bases*, Rio de Janeiro, 1979.

Dahl 1981

Dahl, Veronica, Translating Spanish into Logic through Logic, *Am. Journal of Comp. Linguistics*, 7(3):149-164 (1981).

Dahl 1982

Dahl, Veronica, On Database Systems Development Through Loigc, *ACM Trans. on Database Systems*, 7(1):102-123 (1982).

Dahl and Abramson 1984

Dahl, Veronica, and Abramson, Harvey, On Gapping Grammars, *Proceedings of the Second Intl. Logic Programming Conf.*, Uppsala, Sweden, 1984.

Deliyanni and Kowalski 1979

Deliyanni, A., and Kowalski, R.A., Logic and Semantic Networks, *Comm. ACM*, 22(3):184-192 (1979).

Dietrich and Warren 1986

Dietrich, Susan Wagner and Warren, David S., Extension Tables: Memo Relations in Logic Programming, Technical Report 86/18, C.S. Dept., SUNY, Stony Brook, New York, 1986.

Earley 1970

Earley, Jay, An efficient context-free parsing algorithm, *Comm. ACM* 6(8):451-455 (1970).

Gallaire, et al. 1984

Gallaire, H., Minker, J., Nicolas, J.-M., Logic and Databases: A Deductive Approach, *Computing Surveys*, 16(2):153-185 (1984).

Goguen, et al. 1978

Goguen, J.A., Thatcher, J.W., Wagner, E.G., An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types, in *Current Trends in Programming Methodology*, Vol. 4, Raymond Yeh (ed.), p. 80-149, Prentice-Hall, 1978.

Henschen and Naqvi 1984

Henschen, L.J. and Naqvi, S.A., On Compiling Queries in Recursive First-Order Databases, *J. ACM* 31(1):47-85 (1984).

Hirsh 1986

Hirsh, Susan B., P-PATR: A Compiler for Unification-Based Grammars, M.A. Thesis, Stanford University, Stanford, CA, 1986.

Hirschman and Puder 1986

Hirschman, Lynette, and Puder, Karl, Restriction Grammar: A Prolog Implementation, in *Logic Programming and Its Applications*, Michel van Caneghem and David H.D. Warren (eds.), Ablex, Norwood, NJ, 1986.

Kaplan 1973

Kaplan, Ronald, A General Syntactic Processor, in: Randall Rustin, Ed., *Natural Language Processing*, Algorithmics Press, New York, NY, 1973.

Karttunen 1984

Karttunen, Lauri, Features and Values, *10th Intl. Conf. on Computational Linguistics* (COLING-84), Stanford, CA, 1984.

Karttunen 1986

Karttunen, Lauri, D-PATR: A Development Environment for Unification-Based Grammars, Tech. Report CSLI-86-48, Center for the Study of Language and Information, Stanford, CA, 1986.

Karttunen and Kay 1985

Karttunen, Lauri, and Kay, Martin, Structure Sharing with Binary Trees, *23rd Annual Meeting of the Assoc. for Computational Linguistics*, Chicago, IL, 1985.

Kay 1979

Kay, Martin, Functional Grammar, *Proceedings of the 5th Annual Meeting of the Berkeley Linguistics Society*, Berkeley, CA, 1979.

Kay 1984a

Kay, Martin, Functional Unification Grammar: A Formalism for Machine Translation, *Proc. 22nd Ann. Meeting of the Assoc. for Computational Linguistics* (COLING), Stanford University, Palo Alto, CA, 1984.

Kay 1984b

Kay, Martin, Unification in Grammar, *Natural Lang. Understanding and Logic Programming Conf. Proceedings*, IRISA-INRIA, Rennes, France, 1984.

Kay 1985

Kay, Martin, Parsing in Functional Unification Grammar, in: *Natural Language Parsing*, D.R. Dowty, L.Karttunen, A.M. Zwicky (eds.), Cambridge University Press, Cambridge, 1985.

Knuth 1968

Knuth, Donald E., Semantics of Context Free Languages, *Math. Systems Theory*, 2(2): 127-146 (1968).

Kowalski 1979

Kowalski, R.A., *Logic for Problem Solving*, Elsevier Science Publishing, NY, 1979.

Lipson 1981

Lipson, John D., *Elements of Algebra and Algebraic Computing*, Addison-Wesley, Reading, MA, 1981.

Maier 1980

Maier, David, DAGs as Lattices: Extended Abstract, Unpublished manuscript, 1980.

Maier 1983

Maier, David, *The Theory of Relational Databases*, Computer Science Press, Rockville, Maryland, 1983.

Maier and Warren 1988

Maier, David, Warren, David S., *Computing with Logic: An Introduction to Logic Programming using Prolog*, Benjamin/Cummings Publishing Co., Menlo Park, CA, 1988.

McCord 1980

McCord, Michael C., Slot Grammars, *Computational Linguistics*, 6(1):31-43 (1980).

McCord 1982

McCord, Michael C., Using Slots and Modifiers in Logic Grammars for Natural Language, *Artificial Intelligence*, 18(3):327-368 (1982).

McCord 1984

McCord, Michael C., Semantic Interpretation for the Epistle System, *Proceedings of the Second Intl. Logic Programming Conf.*, Uppsala, Sweden, 1984.

McCord 1985

McCord, Michael C., Modular Logic Grammars, *23rd Annual Meeting of the Assoc. for Computational Linguistics*, Chicago, IL, 1985.

McCord 1986

McCord, Michael C., Focalizers, the Scoping Problem, and Semantic Interpretation Rules in Logic Grammars, in *Logic Programming and Its Applications*, Michel van Caneghem and David H.D. Warren (eds.), Ablex, Norwood, NJ, 1986.

Nilsson 1971

Nilsson, Nils J., *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, 1971.

Pereira 1981

Pereira, F.C.N., Extraposition Grammars, *Computational Linguistics*, 7(4):243-256 (1981).

Pereira 1986

Pereira, F.C.N., A Structure Sharing Representation for Unification-Based Grammar Formalisms, *23rd Annual Meeting of the Assoc. for Computational Linguistics*, Chicago, IL, 1985. Also appears in: Tech. Report CSLI-86-48, Center for the Study of Language and Information, Stanford, CA, 1986.

Pereira 1987

Pereira, F.C.N., Grammars and Logics of Partial Information, *Proceedings of the 4th Intl. Conf. on Logic Programming*, vols 1-2, Melbourne, Australia, J-L Lassez (ed.), MIT Press, 1987.

Pereira and Shieber 1984

Pereira, F.C.N., Shieber, S.M., The Semantics of Grammar Formalisms Seen as Computer Languages, *10th Intl. Conf. on Computational Linguistics* (COLING-84), Stanford, CA, 1984.

Pereira and Warren 1980

Pereira, F.C.N. and Warren, D.H.D., Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence*, 13:231-278 (1980).

Pereira and Warren 1983

Pereira, F.C.N. and Warren D.H.D., Parsing as deduction, *ACL Conference*

Proceedings, 1983.

Porter 1986

Porter, Harry H. III, Earley Deduction, Technical Report CS/E-86-002, Oregon Graduate Center, Beaverton, OR, 1986.

Porter 1987

Porter, Harry H. III, Incorporating Inheritance and Feature Structures into a Logic Grammar Formalism, *25th Annual Meeting of the Assoc. for Computational Linguistics*, Stanford, CA, 1987.

Pullum 1984

Pullum, Geoffrey K., On Two Recent Attempts to Show that English is Not a CFL, *Computational Linguistics*, 10(3-4):182-186 (1984).

Robinson 1965

Robinson, J.A., A Machine-Oriented Logic Based on the Resolution Principle, *J. ACM*, 12(1):23-41 (1965).

Sager 1981

Sager, Naomi, *Natural Language Information Processing*, Addison-Wesley, Reading, MA, 1981.

Shieber 1984

Shieber, Stuart M., The Design of a Computer Language for Linguistic Information, *10th Intl. Conf. on Computational Linguistics* (COLING-84), Stanford, CA, 1984.

Shieber 1985a

Shieber, Stuart M., An Introduction to Unification-Based Approaches to Grammar, Tutorial Session Notes, *23rd Annual Meeting of the Assoc. for Computational Linguistics*, Chicago, IL, 1985.

Shieber 1985b

Shieber, Stuart M., Using Restriction to Extend Parsing Algorithms for Complex-Feature-Bsaed Formalisms, *23rd Annual Meeting of the Assoc. for Computational Linguistics*, Chicago, IL, 1985.

Smith, et al. 1986

Smith, D.E., Genesereth, M.R. and Ginsberg, M.L., Controlling Recursive Inference, *Artificial Intelligence*, 30(3):343-389 (1986).

Stickel 1984

Stickel, Mark E., A Prolog Technology Theorem Prover, *Symposium on Logic Programming*, Atlantic City, NJ, 1984.

Ullman 1984

Ullman, J.D., Testing Applicability of Top-Down Capture Rules, Unpublished memorandum, Dept. of CS, Stanford University, 1984.

Ullman 1985

Ullman, J.D., Implementation of Logical Query Languages for Databases, *ACM Trans. Database Systems*, 10:4 (1985).

Van Emden and Kowalski 1976

Van Emden, M.H., Kowalski, R.A., The Semantics of Predicate Logic as a Programming Language, *J. ACM*, 23(4):733-742 (1976).

Warren 1981

Warren, D.H.D., Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *7th Conf. on Very Large Data Bases*, Cannes, France, 1981.

Warren and Pereira 1982

Warren, D.H.D., Pereira, F.C.N., An Efficient and Easily Adaptable System for Interpreting Natural Language Queries, *Computational Linguistics*, 8(3-4):110-122

(1982).

Winograd 1983

Winograd, Terry, *Language as a Cognitive Process, Vol. 1: Syntax*, Addison-Wesley, Reading, MA, 1983.

Woods 1970

Woods, William A., Transition Network Grammars for Natural Language Analysis, *Comm. ACM.*, 13(10):591-606 (1970).

Appendix 1:
An Example Grammar

This appendix includes a sample IG grammar, called demoGrammar.ig, which is discussed in Chapter 5. This grammar is designed to illustrate the similarity of Definite Clause Grammars and Inheritance Grammars and how easily taxonomic reasoning can be incorporated into an existing DCG by using an IG.

```
/**********  syntax section **********/

s(Predicate) -->
            np(Subject),vp(Subject,Predicate,singular),
            {addFact(Predicate)}.
s(Predicate) -->
            [did],
            np(Subject),
            vp(Subject,Predicate,infinitive),
            {yesNo(Predicate)}.

vp(Subject,Predicate,VForm) -->
            [V],{verb(V,Verb,VForm),
            verbMeaning(Verb,Predicate,
                        [np(Subject)|Complements])},
            vpComplements(Complements).

vpComplements([np(Complement)|Complements]) -->
            np(Complement),
            vpComplements(Complements).
vpComplements([]) --> .
vpComplements([in(Complement)|Complements]) -->
            [in],
            np(Complement),
            vpComplements(Complements).

np(N) --> [N],{properNoun(N)}.
np(Predicate) -->
            [A],{article(A)},[N],{noun(N,Predicate)}.
```

```
dcgConnect(W,[W|L],L).


/**********  semantic processing section  **********/


yesNo(Predicate) :- knowledge(Predicate),
            write(">>> Yes."),nl.
yesNo(Predicate) :- not(knowledge(Predicate)),
            write(">>> No."),nl.

addFact(Predicate) :-
            assert(knowledge(Predicate)),
            write(">>> Okay."),nl,
            generateSentence(Predicate).


/**********  generation grammar  **********/


generateSentence(Predicate) :-
            genS(Predicate,S,[]),
            write (">>>"),writeList(S),write("."),nl.

writeList([]).
writeList([X|L]) :- write(" "),write(X),writeList(L).

genS(Predicate) -->
            {verbMeaning(Verb,Predicate,
                        [np(Subject)|Complements]),
            verb(V,Verb,singular)},
            genNp(Subject),
            [V],
            genComplements(Complements).

genComplements([np(Complement)|Complements]) -->
            genNp(Complement),
            genComplements(Complements).
genComplements([],L,L).
genComplements([in(Complement)|Complements]) -->
            [in],
            genNp(Complement),
            genComplements(Complements).

genNp(N) --> {properNoun(N)},[N].
genNp(Predicate) -->
            {noun(N,Predicate),article(A)},[A],[N].
```

```
/**********  lexicon  **********/


properNoun(person).
properNoun(course).

article(a).

noun(X:grade,X).

verb(gave,give,singular).
verb(give,give,infinitive).

verbMeaning(give,
            take(S,C,F,G),
          [np(F:faculty),np(S:student),
                    np(G:grade),in(C:course)]).


/**********  IS-A hierarchy  **********/


{a,b,c,d,f} < grade.
{maier,kieburtz} < faculty.
{smith,jones} < student.
{student,faculty} < person.
{cs100,cs101} < course.


/**********  knowledge base  **********/


knowledge(take(smith,cs100,kieburtz,a)).
knowledge(take(jones,cs100,kieburtz,c)).
knowledge(take(jones,cs101,maier,d)).
```

Appendix 2: Implementation Size Statistics

The following table lists the source files comprising the two IG interpreter implementations in Smalltalk-80, along with statistics about thier sizes.

File Name	lines	bytes
Inheritance-Grammar.st	3050	110,793
System-Modifications.st	41	1,421
Tokenizer.st	219	8,237
Login.st	3058	104,476
Lattice.st	1845	59,892
totals	8213	284,819

The following table lists the classes comprising the two implementations along with (1) the number of instance variables, (2) the number of class variables, (3) the number of instance messages, and (4) the number of class messages.

class	inst vars	class vars	inst methods	class methods
Lattice	11	3	65	2
LatticeItem	11	0	27	0
LatticeView	0	0	1	1
LatticeController	0	0	16	0
Tokenizer	3	0	7	6
System	0	0	0	3
AR	9	0	18	0
NewProgram	9	0	94	2
NewPsiTerm	3	0	14	0
ParseTerm	4	0	16	4
PsiTerm	4	6	26	10
Literal	3	0	10	3
Clause	2	0	10	2
Program	12	0	51	2
Chunk	6	0	13	1
ChunkModel	2	1	5	3
ChunkView	0	0	1	2
ChunkController	0	2	8	1

Dialog	1	0	2	0
DialogView	0	0	1	2
DialogController	0	2	2	1
LogoView	0	1	1	5
LogoController	2	0	6	0
totals	82	15	394	50

Appendix 3: Comparison of PATR-II and IG by Example

This appendix contains two grammars to illustrate, by example, the similarities and differences between the Inheritance Grammar formalism and the PATR-II formalism. The first is a PATR-II grammar that was copied from the PATR-II literature [Shieber 1985a, Appendix A.3]. The second is an Inheritance Grammar that describes the same target language as the first.

PATR-II Grammar

```
;;; -*- Mode:  PATR -*-

;;;=====================================================
;;;              Demonstration Grammar Three
;;;
;;; Includes:  subject-verb agreement
;;;            complex subcategorization
;;;            logical form construction
;;;=====================================================

Parameter:  Start symbol is S.

Parameter:  Restrictor is <cat>
                          <head form>.

Parameter:  Attribute order is cat lex sense head
                               syncat first rest
                               form agreement
                               person number gender
                               trans pred arg1 arg2
                               s np vp vp_1 vp_2 vp_3 v.

;;;=====================================================
;;;                       Grammar Rules
;;;=====================================================
```

```
Rule {sentence formation}

     S → NP VP:

          <S head> = <VP head>
          <S head form> = finite
          <VP syncat first> = <NP>
          <VP syncat rest> = end.

Rule {trivial verb phrase}

     VP → V:

          <VP head> = <V head>
          <VP syncat> = <V syncat>

Rule {complements}

     VP_1 → VP_2 X:

          <VP_1 head> = <VP_2 head>
          <VP_2 syncat first> = <X>
          <VP_2 syncat rest> = <VP_1 syncat>.

;;;=====================================================
;;;                        Lexicon
;;;=====================================================

Lexicon root.

Word uther:

          <cat> = NP
          <head agreement gender> = masculine
          <head agreement person> = third
          <head agreement number> = singular
          <head trans> = uther.

Word cornwall:

          <cat> = NP
          <head agreement gender> = masculine
          <head agreement person> = third
          <head agreement number> = singular
          <head trans> = cornwall.

Word knights:

          <cat> = NP
```

```
          <head agreement gender> = masculine
          <head agreement person> = third
          <head agreement number> = plural
          <head trans> = knights.

Word sleeps:

          <cat> = V
          <head form> = finite
          <syncat first cat> = NP
          <syncat first head agreement person> = third
          <syncat first head agreement number> = singular
          <syncat rest> = end
          <head trans pred> = sleep
          <head trans arg1> = <syncat first head trans>.

Word sleep:

          <cat> = V
          <head form> = finite
          <syncat first cat> = NP
          <syncat first head agreement number> = plural
          <syncat rest> = end
          <head trans pred> = sleep
          <head trans arg1> = <syncat first head trans>.

Word sleep:

          <cat> = V
          <head form> = nonfinite
          <syncat first cat> = NP
          <syncat rest> = end
          <head trans pred> = sleep
          <head trans arg1> = <syncat first head trans>.

Word storms:

          <cat> = V
          <head form> = finite
          <syncat first cat> = NP
          <syncat rest first cat> = NP
          <syncat rest first head agreement person> =
                                                   third
          <syncat rest first head agreement number> =
                                                singular
          <syncat rest rest> = end
          <head trans pred> = storm
          <head trans arg1> = <syncat rest first head
                                                  trans>
```

```
            <head trans arg2> = <syncat first head trans>.

Word stormed:

            <cat> = V
            <head form> = partparticiple
            <syncat first cat> = NP
            <syncat rest first cat> = NP
            <syncat rest rest> = end
            <head trans pred> = storm
            <head trans arg1> = <syncat rest first head
                                                    trans>
            <head trans arg2> = <syncat first head trans>.

Word storm:

            <cat> = V
            <head form> = nonfinite
            <syncat first cat> = NP
            <syncat rest first cat> = NP
            <syncat rest rest> = end
            <head trans pred> = storm
            <head trans arg1> = <syncat rest first head
                                                    trans>
            <head trans arg2> = <syncat first head trans>.

Word has:

            <cat> = V
            <head form> = finite
            <syncat first cat> = VP
            <syncat first head form> = pastparticiple
            <syncat first syncat rest> = end
            <syncat first syncat first> = <syncat rest
                                                    first>
            <syncat rest first cat> = NP
            <syncat rest first head agreement number> =
                                                  singular
            <syncat rest first head agreement person> =
                                                     third
            <syncat rest rest> = end
            <head trans pred> = perfective
            <head trans arg1> = <syncat first head trans>.

Word have:

            <cat> = V
            <head form> = finite
            <syncat first cat> = VP
```

```
        <syncat first head form> = pastparticiple
        <syncat first syncat rest> = end
        <syncat first syncat first> = <syncat rest
                                            first>
        <syncat rest first cat> = NP
        <syncat rest first head agreement number> =
                                            plural
        <syncat rest rest> = end
        <head trans pred> = perfective
        <head trans arg1> = <syncat first head trans>.
```

Word persuades:

```
        <cat> = V
        <head form> = finite
        <syncat first cat> = NP
        <syncat rest first cat> = VP
        <syncat rest first head form> = infintival
        <syncat rest first syncat rest> = end
        <syncat rest first syncat first> = <syncat
                                            first>
        <syncat rest rest first cat> = NP
        <syncat rest rest first head agreement
                                number> = singular
        <syncat rest rest first head agreement
                                person> = third
        <syncat rest rest rest> = end
        <head trans pred> = persuade
        <head trans arg1> = <syncat rest rest first
                                        head trans>
        <head trans arg2> = <syncat first head trans>
        <head trans arg3> = <syncat rest first head
                                            trans>.
```

Word to:

```
        <cat> = V
        <head form> = infinitival
        <syncat first cat> = VP
        <syncat first head form> = nonfinite
        <syncat first syncat rest> = end
        <syncat first syncat first> = <syncat rest
                                            first>
        <syncat rest first cat> = NP
        <syncat rest rest> = end
        <head trans> = <syncat first head trans>.
```

Inheritance Grammar

```
/*******************************************************/
/*                 Grammar Rules                        */
/*******************************************************/

s(head=>H:(form=>finite)) -->
        N:np,
        vp(head=>H;syncat=>[N]).

vp(head=>H;syncat=>S) -->
        v(head=>H;syncat=>S).

vp(head=>H;syncat=>S) -->
        vp(head=>H;syncat=>[X|S]),
        X.


/*******************************************************/
/*                    Lexicon                          */
/*******************************************************/


/*****  word: uther  *****/

np(head=>(
      agreement=>(
              gender=>masculine;
              person=>third;
              number=>singular);
      trans=>uther))
  --> [uther].


/*****  word: cornwall  *****/

np(head=>(
      agreement=>(
             gender=>masculine;
             person=>third;
             number=>singular);
      trans=>cornwall))
  --> [cornwall].


/*****  word: knights  *****/

np(head=>(
```

```
      agreement=>(
            gender=>masculine;
            person=>third;
            number=>plural);
      trans=>knights))
  --> [knights].


/*****  word: sleeps  *****/

v(    head=>(
            form=>finite;
            trans=>sleep(X));
      syncat=>[
            np( head=>(
                  agreement=>(
                        person=>third;
                        number=>singular);
                  trans=>X))])
  --> [sleeps].


/*****  word: sleep  *****/

v(    head=>(
            form=>finite;
            trans=>sleep(Subj));
      syncat=>[
            np( head=>(
                  agreement=>(
                       number=>plural);
                  trans=>Subj))])
  --> [sleep].


/*****  word: sleep  *****/

v(    head=>(
            form=>nonfinite;
            trans=>sleep(Subj));
      syncat=>[
            np( head=>(
                  trans=>Subj))])
  --> [sleep].


/*****  word: storms  *****/

v(    head=>(
```

```
            form=>finite;
            trans=>storm(Subj,Obj));
      syncat=>[
            np(head=>(
                  trans=>Obj)),
            np(head=>(
                  agreement=>(
                        person=>third;
                        number=>singular);
                  trans=>Subj))])
  --> [storms].


/*****  word: stormed  *****/

v(    head=>(
            form=>presentparticiple;
            trans=>storm(Subj,Obj));
      syncat=>[
            np( head=>(
                  trans=>Obj)),
            np( head=>(
                  trans=>Subj))])
  --> [stormed].


/*****  word: storm  *****/

v(    head=>(
            form=>nonfinite;
            trans=>storm(Subj,Obj));
      syncat=>[
            np( head=>(
                  trans=>Obj)),
            np( head=>(
                  trans=>Subj))])
  --> [storm].


/*****  word: has  *****/

v(    head=>(
            form=>finite;
            trans=>perfective(Pred));
      syncat=>[
            vp(   head=>(
                        form=>presentparticiple;
                        trans=>Pred);
                  syncat=>[S]),
```

```
            S:np( head=>(
                        agreement=>(
                              number=>singular;
                              person=>third)))
            ])
  --> [has].


/*****  word: have  *****/

v(     head=>(
             form=>finite;
             trans=>perfective(P));
      syncat=>[
             vp(   head=>(
                         form=>presentparticiple;
                         trans=>P);
                   syncat=>[N]),
             N:np(
                   head=>(
                         agreement=>(
                               number=>plural)))])
  --> [have].


/*****  word: persuades  *****/

v(     head=>(
             form=>finite;
             trans=>persuade(S,O,P));
      syncat=>[
             N:np( head=>(
                         trans=>O)),
             vp(   head=>(
                         form=>infinitival;
                               trans=>P);
                         syncat=>[N]),
             np(   head=>(
                         agreement=>(
                               number=>singular;
                               person=>third);
                         trans=>S))])
  --> [persuades].


/*****  word: to  *****/

v(     head=>(
             form=>infinitival;
```

```
          trans=>P);
     syncat=>[
          vp(    head=>(
                         form=>nonfinite;
                         trans=>P);
                 syncat=>[N]),
          N:np])
--> [to].
```

Appendix 4: An Extended Inheritance Grammar

This appendix includes an example Inheritance Grammar, followed by examples of sentences it can parse and the corresponding output produced by the grammar. This grammar took 86 seconds to compile using the Inheritance Grammar compilation system discussed in Chapter 5 on a Tektronix 4317.

The lexical transformation done by the compilation system is slightly different than the transformation described in Chapter 5 for the interpreter system. In particular, the `parse it` message transforms an input symbol (i.e., token) such as `teaches` into the symbol `lexTEACHES`. The changed transformation serves two functions. First, the grammar is robust against capitalization errors in the input. Second, by prefixing `lex` to all symbols that are matched against input tokens, it makes the grammar a little more readable.

The grammar first translates the input into a database query and then executes that query against a small database, which is included directly in the grammar. The query contains information to verify some preconceptions discovered in the input. For example, if a noun phrase in the input is in plural form, the query contains a different quantifier than if the same noun phrase had appeared in singular form. During the execution of the query, these preconditions are checked and, if they are not satisfied, the text `Your input makes an invalid assumption.` is printed to alert the user that the answer (which is also printed) may be incorrect or misleading. Such a response is exhibited in the input-response pairs that follow the grammar.

The Grammar

```
/******************** DIALOG CONTROL ********************/

   parse -->
      sentence (translation=>R:query),
      { nl, write("TRANSLATION:"), nl,
      write (R), nl, nl,
      R, nl}.

   parseNP -->
      NP:nounPhr,
      { nl, write("TRANSLATION:"), nl,
      write (NP), nl, nl }.

{ yesNo,declare,howMany,whoOrWhat,listWhoOrWhat,which } < query.



/******************** SEMANTIC TAXONOMY ********************/

{ thing, event, time } < semanticEntity.

{ animate, inanimate } < thing.

{ male, female, person } < animate.
{ student, faculty } < person.
{ harryPorter, marySmith, johnSmith } < student.
{ davidMaier, dickKieburtz, janeBrown } < faculty.
{ harryPorter, johnSmith, davidMaier, dickKieburtz } < male.
{ marySmith, janeBrown } < female.

{ dept, course, grade } < inanimate.
cs < dept.
{ cs100, cs101, cs102 } < course.
{ a, b, c, d, f } < grade.

year < time.
{ 1986, 1987, 1988 } < year.

offering < event.
{ cs100a, cs100b, cs101a, cs102a, cs102b } < offering.



/******************** QUANTIFIER TAXONOMY ********************/
```

```
{ question, nonQuestion } < quantifier.
{ howMany, which, whoOrWhat } < question.
{ every, a, any, definite } < nonQuestion.
{ defSingular, defPlural } < definite.


/*******************  SEMANTIC DATABASE  *******************/

nameR (id=>harryPorter, first=>lexHARRY, last=>lexPORTER).
nameR (id=>marySmith, first=>lexMARY, last=>lexSMITH).
nameR (id=>johnSmith, first=>lexJOHN, last=>lexSMITH).
nameR (id=>davidMaier, first=>lexDAVID, last=>lexMAIER).
nameR (id=>dickKieburtz, first=>lexDICK, last=>lexKIEBURTZ).
nameR (id=>janeBrown, first=>lexJANE, last=>lexBROWN).

courseR (course=>cs100,
         dept=>cs,
         subject=>lexCS,
         number=>lex100,
         name=>[lexINTRO, lexTO, lexCOMPUTER, lexSCIENCE]).
courseR (course=>cs101,
         dept=>cs,
         subject=>lexCS,
         number=>lex101,
         name=>[lexCOMPILER, lexDESIGN]).
courseR (course=>cs102,
         dept=>cs,
         subject=>lexCS,
         number=>lex102,
         name=>[lexARTIFICIAL, lexINTELLIGENCE]).

deptR (dept=>cs, name=>[lexCOMPUTER, lexSCIENCE]).

gradeR (entity=>a, lex=>lexA).
gradeR (entity=>b, lex=>lexB).
gradeR (entity=>c, lex=>lexC).
gradeR (entity=>d, lex=>lexD).
gradeR (entity=>f, lex=>lexF).

yearR (entity=>1986, lex=>lex1986).
yearR (entity=>1987, lex=>lex1987).
yearR (entity=>1988, lex=>lex1988).

offeringR (course=>cs100,
           offering=>cs100a,
           faculty=>davidMaier,
           year=>1987).
offeringR (course=>cs100,
```

```
           offering=>cs100b,
           faculty=>dickKieburtz,
           year=>1988).
offeringR (course=>cs101,
           offering=>cs101a,
           faculty=>dickKieburtz,
           year=>1987).
offeringR (course=>cs102,
           offering=>cs102a,
           faculty=>janeBrown,
           year=>1987).
offeringR (course=>cs102,
           offering=>cs102b,
           faculty=>davidMaier,
           year=>1988).

enrollR (offering=>cs100a, student=>harryPorter, grade=>a).
enrollR (offering=>cs101a, student=>harryPorter, grade=>b).
enrollR (offering=>cs102b, student=>harryPorter, grade=>c).
enrollR (offering=>cs100b, student=>marySmith, grade=>c).
enrollR (offering=>cs101a, student=>marySmith, grade=>b).
enrollR (offering=>cs102a, student=>marySmith, grade=>a).
enrollR (offering=>cs100b, student=>johnSmith, grade=>d).
enrollR (offering=>cs101a, student=>johnSmith, grade=>f).
enrollR (offering=>cs102b, student=>johnSmith, grade=>c).

inanimateThingR (id=>Grade) :- gradeR (entity=>Grade).
inanimateThingR (id=>Course) :- courseR (course=>Course).
inanimateThingR (id=>Offering) :- offeringR (offering=>Offering).


/******************  SENTENCE PARSING RULES  ******************/

----------  Parses:  AUX NP VERB ... ?

sentence (translation=>yesNo(Trans)) -->
    [Aux],
    { verbForm (lex=>Aux, root=>R1, tense=>T1, number=>N),
      finiteAux (R1, T1, T2) },
    !,
    nounPhr (result=>NP:(number=>N)),
    { notQuestNP(NP) },
    [Verb],
    { verbForm (lex=>Verb, root=>R2, tense=>T2),
      verbMeaning (root=>R2, predicate=>P,
                   subject=>S, slotList=>SL) },
    verbComplements (subject=>NP, predicate=>P, predSubject=>S,
                     slotList=>SL, holdIn=>nil, holdOut=>nil,
```

```
                    translation=>Trans),
    ['lex?'].

notQuestNP(nounPhrase(quantifier=>question)) :- !, fail.
notQuestNP(X).

---------- Parses:  Every other sentence form

sentence (translation=>Trans) -->
    nounPhr (result=>NP),
    restS (topic=>NP, translation=>Trans).

---------- Parses:  WHO/WHAT  IS/ARE/WAS/WERE  NP?

restS (topic=>NP:(quantifier=>whoOrWhat),
       translation=>listWhoOrWhat(X,P,true)) -->
    [Aux],
    { verbForm (lex=>Aux, root=>be, tense=>T, number=>N),
      finiteAux (be, T, T2) },
    nounPhr (result => (number=>N, quantifier=>nonQuestion,
                        variable=>X, predicate=>P)),
    ['lex?'], !.

---------- Parses:  QUEST-NP VERB ... ?
----------     or:  QUEST-NP AUX NP VERB ... ?

restS (topic=>NP:(quantifier=>question), translation=>Trans) -->
    !,
    questionS (topic=>NP, translation=>Trans),
    ['lex?'].

---------- Parses:  NP VERB ...

restS (topic=>NP:(number=>N), translation=>declare(Trans)) -->
    verbSequence (root=>R, number=>N, tense=>T),
    { verbMeaning (root=>R, predicate=>P,
                   subject=>S, slotList=>SL) },
    verbComplements (subject=>NP, predicate=>P, predSubject=>S,
                     slotList=>SL, holdIn=>nil, holdOut=>nil,
                     translation=>Trans),
    ['lex.'].

---------- Parses:  QUEST-NP VERB ... ?

questionS (topic => NP: (number=>N), translation => Trans)  -->
    verbSequence (root=>R, number=>N, tense=>T),
    { verbMeaning (root=>R, predicate=>P,
                   subject=>S, slotList=>SL) },
    verbComplements (subject=>NP, predicate=>P, predSubject=>S,
```

```
                    slotList=>SL, holdIn=>nil, holdOut=>nil,
                    translation=>Trans).

----------  Parses:  QUEST-NP AUX NP VERB ... ?

questionS (topic => QNP: (number=>N), translation => Trans)  -->
    [Aux], { verbForm (lex=>Aux, root=>R1, tense=>T1, number=>N),
             finiteAux (R1, T1, T2) },
    nounPhr (result => NP2),
    [Verb],
    { verbForm (lex=>Verb, root=>R2, tense=>T2),
      verbMeaning (root=>R2, predicate=>P,
                   subject=>S, slotList=>SL) },
    verbComplements (subject=>NP2, predicate=>P, predSubject=>S,
                    slotList=>SL, holdIn=>QNP, holdOut=>nil,
                    translation=>Trans).



----------  Parses:  IS TEACHING,  HAS TAUGHT,  DID TEACH

verbSequence (root=>R, tense=>T2, number=>N) -->
    [Verb1],
    { verbForm (lex=>Verb1, root=>R1, tense=>T1, number=>N),
               finiteAux (R1, T1, T2) },
    [Verb2],
    { verbForm (lex=>Verb2, root=>R, tense=>T2) }, !.

----------  Parses:  TEACHES,  TAUGHT

verbSequence (root=>R, tense=>T, number=>N) -->
    [Verb], { verbForm (lex=>Verb, root=>R, tense=>T, number=>N),
              presentOrPast (T) }.

presentOrPast (present).
presentOrPast (past).



/*******************  VERB COMPLEMENTS  *******************/

verbComplements ( subject=> nounPhrase (quantifier=>Q,
                                        variable=>X,
                                        number=>N,
                                        predicate=>P),
                  predicate=>Pred1,
                  predSubject=>X,
                  slotList=>SL,
                  holdIn=>Hin,
```

```
                holdOut=>Hout,
                translation=>Pred3) -->
        scanComplements ( slotList=>SL,
                          predicateIn=>Pred1,
                          predicateOut=>Pred2,
                          holdIn=>Hin,
                          holdOut=>Hout),
        { combineQuants ( quantifier=>Q,
                          number=>N,
                          variable=>X,
                          predicate=>P,
                          predicateIn=>Pred2,
                          predicateOut=>Pred3) }.


/*****************  NOUN PHRASE PARSING RULES  *****************/

nounPhr (result=>T) -->
    nounPhr0 (holdIn=>nil, holdOut=>nil, result=>T).

----------

nounPhr0 (holdIn=>Hin,
          holdOut=>Hout,
          result=>Result) -->
    [Det],
    determiner ( lex=>Det, number=>N, quantifier=>Q),
    [HeadNoun],
    { nounForm (lex=>HeadNoun, number=>N, concept=>X),
      nounMeaning (concept=>X, pred=>P1, slotList=>SL) },
    scanComplements ( slotList=>SL,
                      predicateIn=>P1,
                      predicateOut=>P2,
                      holdIn=>Hin,
                      holdOut=>Hout),
    relClause ( nounPhrIn=>nounPhrase ( number=>N,
                                        quantifier=>Q,
                                        variable=>X,
                                        predicate=>P2),
                nounPhrOut=>Result).

nounPhr0 ( holdIn=>Hin,
           holdOut=>Hout,
           result=>nounPhrase ( number=>N,
                                quantifier=>Q,
                                variable=>X,
                                predicate=>P)) -->
    [Word],
```

```
    { pronoun ( lex=>Word,
                quantifier=>Q,
                number=>N,
                quantifiedEntity=>X,
                predicate=>P) }.

nounPhr0 ( holdIn=>Hin,
           holdOut=>Hout,
           result=>NP) -->
    [Word],
    properNP ( word=>Word, entity=>X, predicate=>P),
    nounPhr1 ( holdIn=>Hin,
               holdOut=>Hout,
               resultIn=>nounPhrase (number=>singular,
                                     quantifier=>defSingular,
                                     variable=>X,
                                     predicate=>P),
               resultOut=>NP).

----------

nounPhr1 (holdIn=>H,holdOut=>H,resultIn=>NP,resultOut=>NP) --> .

nounPhr1 (holdIn=>Hin,
          holdOut=>Hout,
          resultIn=>nounPhrase (variable=>Owner),
          resultOut=>nounPhrase (number=>N,
                                 quantifier=>DefQuant,
                                 variable=>X,
                                 predicate=>Pred2)) -->
    ['lex''S', HeadNoun],
    { nounForm (lex=>HeadNoun, number=>N, concept=>X),
      defQuantifier (N, DefQuant),
      nounMeaning (concept=>X,
                   pred=>Pred1,
                   slotList=>[npPossObj(object=>Owner) | SL] ) },
    scanComplements ( slotList=>SL,
                      predicateIn=>Pred1,
                      predicateOut=>Pred2,
                      holdIn=>Hin,
                      holdOut=>Hout).

defQuantifier (plural, defPlural).
defQuantifier (singular, defSingular).


/********************  PROPER NOUN PHRASES  ********************/
```

```
properNP (word=>W, entity=>P:person, predicate=>nameR(id=>P)) -->
    name(word=>W,entity=>P),!.
properNP (word=>W, entity=>C:course, predicate=>P) -->
    [N],
    { P:courseR(course=>C,subject=>W,number=>N) }, !.
properNP (word=>W, entity=>C:course, predicate=>P) -->
    { P:courseR(course=>C,name=>[W|L]) }, scanList(L), !.
properNP (word=>W, entity=>Y:year, predicate=>P) -->
    { P:yearR(entity=>Y,lex=>W) }, !.
properNP (word=>W, entity=>D:dept, predicate=>P) -->
    { P:deptR(dept=>D,name=>[W|L]) }, scanList(L), !.

scanList ([]) -->.
scanList ([W|L]) --> [W], scanList (L).

name (word=>lexPROF, entity=>ID:faculty) -->
    [L], !, { nameR(id=>ID,last=>L) }.
name (word=>lexMR, entity=>ID:student) -->
    [L], !, { nameR(id=>ID:male,last=>L) }.
name (word=>lexMS, entity=>ID:student) -->
    [L], !, { nameR(id=>ID:female,last=>L) }.
name (word=>L, entity=>ID:person) -->
    { nameR(id=>ID,last=>L) }.
name (word=>F, entity=>ID:person) -->
    [L], { nameR(id=>ID,first=>F,last=>L)} , ! .



/******************** RELATIVE CLAUSES ********************/

relClause (nounPhrIn=>NP, nounPhrOut=>NP) --> .

relClause (nounPhrIn=>H1: (quantifier=>Q,
                           number=>N,
                           variable=>X,
                           predicate=>P0),
           nounPhrOut=>nounPhrase (
                           quantifier=>Q2,
                           number=>N,
                           variable=>X,
                           predicate=>Q3:(X, and(P0,P2), true)))
-->
    relPronoun (X),
    possiblyHeldNounPhr (holdIn=>H1,
                         holdOut=>H2,
                         result=>NP: (number=>N2)),
    verbSequence (root=>R, number=>N2, tense=>T),
    { verbMeaning (root=>R, predicate=>P1,
                   subject=>S, slotList=>SL) },
```

```
    verbComplements (subject=>NP,
                     predicate=>P1,
                     predSubject=>S,
                     slotList=>SL,
                     holdIn=>H2,
                     holdOut=>nil,
                     translation=>P2),
    { copy (Q,Q2), copy(Q,Q3) }.



relPronoun            --> [lexTHAT].
relPronoun (animate) --> [lexWHO].
relPronoun            --> [lexWHICH].



/********************  COMPLEMENTS  ********************/

scanComplements ( slotList=>[],
                  predicateIn=>P,
                  predicateOut=>P,
                  holdIn=>H,
                  holdOut=>H) --> !.
scanComplements ( slotList=>[S|SL],
                  predicateIn=>P1,
                  predicateOut=>P3,
                  holdIn=>H1,
                  holdOut=>H3) -->
     S:slot (holdIn=>H1,
             holdOut=>H2,
             predicateIn=>P1,
             predicateOut=>P2),
     scanComplements (slotList=>SL,
                      predicateIn=>P2,
                      predicateOut=>P3,
                      holdIn=>H2,
                      holdOut=>H3).


{ prepObject, npPossObj, directObject } < slot.


prepObject (optional=>yes, holdIn=>H, holdOut=>H,
            predicateIn=>P, predicateOut=>P) -->.
prepObject (prep=>Word, object=>X, holdIn=>Hin, holdOut=>Hout,
            predicateIn=>PredIn, predicateOut=>PredOut) -->
     [Word],
     PN:possiblyHeldNounPhr (
```

```
                  holdIn=>Hin,
                  holdOut=>Hout,
                  result=>nounPhrase (quantifier=>Q,
                                      number=>N,
                                      variable=>X, predicate=>P)),
    { combineQuants (quantifier=>Q,
                     number=>N,
                     variable=>X,
                     predicate=>P,
                     predicateIn=>PredIn,
                     predicateOut=>PredOut)}.



npPossObj (object=>X, holdIn=>Hin, holdOut=>Hout,
           predicateIn=>PredIn, predicateOut=>PredOut) -->
    [lexOF],
    possiblyHeldNounPhr ( holdIn=>Hin,
                          holdOut=>Hout,
                          result=>nounPhrase (
                                      quantifier=>defSingular,
                                      number=>singular,
                                      variable=>X, predicate=>P)),
    { combineQuants (quantifier=>defSingular,
                     number=>singular,
                     variable=>X,
                     predicate=>P,
                     predicateIn=>PredIn,
                     predicateOut=>PredOut) }.



directObject (object=>X, holdIn=>Hin, holdOut=>Hout,
              predicateIn=>PredIn, predicateOut=>PredOut) -->
    possiblyHeldNounPhr ( holdIn=>Hin,
                          holdOut=>Hout,
                          result=>nounPhrase ( quantifier=>Q,
                                               number=>N,
                                               variable=>X,
                                               predicate=>P)),
    {combineQuants (quantifier=>Q,
                    number=>N,
                    variable=>X,
                    predicate=>P,
                    predicateIn=>PredIn,
                    predicateOut=>PredOut)}.
```

```
possiblyHeldNounPhr (holdIn=>NP, holdOut=>nil, result=>NP) -->
    {notNil (NP)}.
possiblyHeldNounPhr (holdIn=>H, holdOut=>H, result=>R) -->
    nounPhr (result=>R).


notNil (nil) :- !, fail.
notNil (X).



--   Note: "predicateIn=>" and "quantifier=>" must be bound
--         upon calling this predicate.
combineQuants ( quantifier=>Q,
                variable=>X,
                predicate=>P1,
                predicateIn=>howMany (Y;P2;P3),
                predicateOut=>howMany (Y;P2;Q2:(X;P1;P3))) :-
    !, copy(Q,Q2).
combineQuants ( quantifier=>Q,
                variable=>X,
                predicate=>P1,
                predicateIn=>which (N;Y;P2;P3),
                predicateOut=>which (N;Y;P2;Q2:(X;P1;P3))) :-
    !, copy(Q,Q2).
combineQuants ( quantifier=>which,
                number=>N,
                variable=>X,
                predicate=>P1,
                predicateIn=>P2,
                predicateOut=>which (N;X;P1;P2)) :-
    !.
combineQuants ( quantifier=>Q,
                variable=>X,
                predicate=>P1,
                predicateIn=>P2,
                predicateOut=>Q2:(X;P1;P2)) :-
    copy(Q,Q2).


/********************  COMMON NOUNS  ********************/

nounForm (lex=>lexCOURSE, concept=>course, number=>singular).
nounForm (lex=>lexCOURSES, concept=>course, number=>plural).
nounForm (lex=>lexPERSON, concept=>person, number=>singular).
nounForm (lex=>lexPERSONS, concept=>person, number=>plural).
nounForm (lex=>lexSTUDENT, concept=>student, number=>singular).
nounForm (lex=>lexSTUDENTS, concept=>student, number=>plural).
```

```
nounForm (lex=>lexPROFESSOR, concept=>faculty, number=>singular).
nounForm (lex=>lexPROFESSORS, concept=>faculty, number=>plural).
nounForm (lex=>lexA, concept=>a, number=>singular).
nounForm (lex=>lexB, concept=>b, number=>singular).
nounForm (lex=>lexC, concept=>c, number=>singular).
nounForm (lex=>lexD, concept=>d, number=>singular).
nounForm (lex=>lexE, concept=>f, number=>singular).
nounForm (lex=>lexGRADE, concept=>grade, number=>singular).
nounForm (lex=>lexGRADES, concept=>grade, number=>plural).
nounForm (lex=>lexTHING, concept=>inanimate, number=>singular).
nounForm (lex=>lexTHINGS, concept=>inanimate, number=>plural).

nounMeaning ( concept => G:grade,
              pred => gradeR(entity=>G),
              slotList => [] ).
nounMeaning ( concept => G:grade,
              pred => and (enrollR (offering=>O:offering,
                                    student=>S:student,
                                    grade=>G:grade),
                           offeringR (course=>C:course,
                                      offering=>O) ),
              slotList => [ npPossObj(object=>S),
                            prepObject(prep=>lexIN,
                                       object=>C,
                                       optional=>no) ] ).
nounMeaning ( concept => P:person,
              pred => nameR(id=>P),
              slotList => [] ).
nounMeaning ( concept => C:course,
              pred => courseR (course=>C:course, dept=>D:dept),
              slotList => [ prepObject (prep=>lexIN,
                                        object=>D,
                                        optional=>yes) ] ).


/*****************  PRONOUNS AND DETERMINERS  *****************/

pronoun (lex=>lexWHAT,
         quantifier=>which,
         quantifiedEntity=>X:inanimate,
         predicate=>inanimateThingR(id=>X)).
pronoun (lex=>lexWHAT,
         quantifier=>whoOrWhat,
         quantifiedEntity=>X:inanimate,
         predicate=>inanimateThingR(id=>X)).
pronoun (lex=>lexWHO,
         quantifier=>which,
         quantifiedEntity=>X:animate,
```

```
          predicate=>nameR(id=>X)).
pronoun (lex=>lexWHO,
          quantifier=>whoOrWhat,
          quantifiedEntity=>X:animate,
          predicate=>nameR(id=>X)).
pronoun (lex=>lexSOMEBODY,
          quantifier=>indefinite,
          quantifiedEntity=>X:animate,
          number=>singular).
pronoun (lex=>lexEVERYBODY,
          quantifier=>every,
          quantifiedEntity=>X:animate,
          number=>singular).



determiner (lex=>lexTHE,
            number=>singular,
            quantifier=>defSingular) --> .
determiner (lex=>lexTHE,
            number=>plural,
            quantifier=>defPlural) --> .
determiner (lex=>lexA,
            number=>singular,
            quantifier=>a) --> .
determiner (lex=>lexAN,
            number=>singular,
            quantifier=>a) --> .
determiner (lex=>lexANY,
            quantifier=>any) --> .
determiner (lex=>lexHOW,
            number=>plural,
            quantifier=>howMany) --> [lexMANY].
determiner (lex=>lexEVERY,
            number=>singular,
            quantifier=>every) --> .
determiner (lex=>lexWHAT,
            quantifier=>which) --> .
determiner (lex=>lexWHICH,
            quantifier=>which) --> .


/********************  VERBS  ********************/

finiteAux (do, present, infinitive).
finiteAux (be, present, presentParticiple).
finiteAux (have, present, pastParticiple).
finiteAux (do, past, infinitive).
```

```
finiteAux (be, past, presentParticiple).
finiteAux (have, past, pastParticiple).

verbForm (lex=>lexDOES, root=>do,
          tense=>present,
          number=>singular).
verbForm (lex=>lexDO, root=>do,
          tense=>present, number=>plural).
verbForm (lex=>lexDID, root=>do,
          tense=>past).
verbForm (lex=>lexDO, root=>do,
          tense=>infinitive).
verbForm (lex=>lexDONE, root=>do,
          tense=>pastParticiple).
verbForm (lex=>lexDOING, root=>do,
          tense=>presentParticiple).

verbForm (lex=>lexIS, root=>be,
          tense=>present,
          number=>singular).
verbForm (lex=>lexARE, root=>be,
          tense=>present, number=>plural).
verbForm (lex=>lexWAS, root=>be,
          tense=>past, number=>singular).
verbForm (lex=>lexWERE, root=>be,
          tense=>past, number=>plural).
verbForm (lex=>lexBE, root=>be,
          tense=>infinitive).
verbForm (lex=>lexBEEN, root=>be,
          tense=>pastParticiple).
verbForm (lex=>lexBEING, root=>be,
          tense=>presentParticiple).

verbForm (lex=>lexHAS, root=>have,
          tense=>present, number=>singular).
verbForm (lex=>lexHAVE, root=>have,
          tense=>present, number=>plural).
verbForm (lex=>lexHAD, root=>have,
          tense=>past).
verbForm (lex=>lexHAVE, root=>have,
          tense=>infinitive).
verbForm (lex=>lexHAVEN, root=>have,
          tense=>pastParticiple).
verbForm (lex=>lexHAVING, root=>have,
          tense=>presentParticiple).

verbForm (lex=>lexTEACHES, root=>teach,
          tense=>present, number=>singular).
verbForm (lex=>lexTEACH, root=>teach,
```

```
          tense=>present, number=>plural).
verbForm (lex=>lexTAUGHT, root=>teach,
          tense=>past).
verbForm (lex=>lexTEACH, root=>teach,
          tense=>infinitive).
verbForm (lex=>lexTAUGHT, root=>teach,
          tense=>pastParticiple).
verbForm (lex=>lexTEACHING, root=>teach,
          tense=>presentParticiple).

verbForm (lex=>lexTAKES, root=>take,
          tense=>present, number=>singular).
verbForm (lex=>lexTAKE, root=>take,
          tense=>present, number=>plural).
verbForm (lex=>lexTOOK, root=>take,
          tense=>past).
verbForm (lex=>lexTAKE, root=>take,
          tense=>infinitive).
verbForm (lex=>lexTAKEN, root=>take,
          tense=>pastParticiple).
verbForm (lex=>lexTAKING, root=>take,
          tense=>presentParticiple).

verbForm (lex=>lexGETS, root=>get,
          tense=>present, number=>singular).
verbForm (lex=>lexGET, root=>get,
          tense=>present, number=>plural).
verbForm (lex=>lexGOT, root=>get,
          tense=>past).
verbForm (lex=>lexGET, root=>get,
          tense=>infinitive).
verbForm (lex=>lexGOTTEN, root=>get,
          tense=>pastParticiple).
verbForm (lex=>lexGETTING, root=>get,
          tense=>presentParticiple).

/* faculty teaches course [in year]. */
verbMeaning ( root=>teach,
              predicate => offeringR (course=>C,
                                      faculty=>F,
                                      year=>Y),
              subject => F:faculty,
              slotList => [ directObject (object=>C:course),
                            prepObject (prep=>lexIN,
                            object=>Y:year,
                            optional=>yes)]).

/* student takes course [from faculty] [in year]. */
verbMeaning ( root=>take,
```

```
                predicate => and (enrollR (student=>S,offering=>O),
                                  offeringR (offering=>O,
                                             course=>C,
                                             faculty=>F,
                                             year=>Y)),
                subject => S:student,
                slotList => [ directObject (object=>C:course),
                              prepObject (prep=>lexFROM,
                                          object=>F:faculty,
                                          optional=>yes),
                              prepObject (prep=>lexIN,
                                          object=>Y:year,
                                          optional=>yes)]).

/* student gets grade [from faculty] [in course] [in year]. */
verbMeaning ( root=>get,
                predicate => and ( enrollR (student=>S,
                                            offering=>O,
                                            grade=>G),
                                   offeringR (offering=>O,
                                              course=>C,
                                              faculty=>F,
                                              year=>Y)),
                subject => S:student,
                slotList => [ directObject(object=>G:grade),
                              prepObject (prep=>lexFROM,
                                          object=>F:faculty,
                                          optional=>yes),
                              prepObject (prep=>lexIN,
                                          object=>C:course,
                                          optional=>yes),
                              prepObject (prep=>lexIN,
                                          object=>Y:year,
                                          optional=>yes)]).

/* student has faculty [for course] [in year]. */
verbMeaning ( root=>have,
                predicate => and (enrollR (student=>S,offering=>O),
                                  offeringR (offering=>O,
                                             course=>C,
                                             faculty=>F,
                                             year=>Y)),
                subject => S:student,
                slotList => [ directObject (object=>F:faculty),
                              prepObject (prep=>lexFOR,
                                          object=>C:course,
                                          optional=>yes),
                              prepObject (prep=>lexIN,
                                          object=>Y:year,
```

```
                                          optional=>yes)]).


/***************  UTILITY OPERATORS  ********************/

and (P, Q) :- P, Q.

not (P) :- P, !, fail.
not (P).

true.

copy (X,Y) :- solutions (X,true,[Y]).

eq (X,X).

dcgConnect(W,[W|L],L).



/***************  QUANTIFIERS  ********************/

every (X, P, Q) :-
    solutions (X, P, L),
    checkAll (X, L, Q).

a (X, P, Q) :-
    P,
    Q,
    !.

any (X, P, Q) :-
    P,
    Q,
    !.

--  defSingular (X, P, Q)
--      P should have exactly one solution.  Find that solution
--      and the binding for X which it imposes.  Ensure that,
--      under this binding for X, Q is also satisfied.  Succeed
--      iff Q succeeds.  On success however, do not bind X,P,
--      or Q.
defSingular (X, P, Q) :-
    solutions (X, P, L),
    presupposeSingleton (L),
    checkAll (X, L, Q), !.

--  defPlural (X, P, Q)
```

```
--      P should have at least one solution.  Find those
--      solutions and the bindings for X they impose.  Ensure
--      that, under each such binding for X, Q is also satisfied.
--      Succeed iff Q succeeds for all bindings.  On success
--      however, do not bind X,P, or Q.
defPlural (X, P, Q) :-
    solutions (X, P, L),
    presupposeNotEmpty (L),
    checkAll (X, L, Q), !.

--  checkAll (X, [...,Yi,... ], P)
--      P will usually be a predicate containing occurrences of
--      X.  Bind X to a Yi in the list.  For each, see if P is
--      then satisfied.  Succeed exactly once iff P is satisfied
--      under all bindings.  On success however, do not bind
--      X,Yi, or P.
checkAll (X, [], Q).
checkAll (X, [Y|L], Q) :-
    check1 (X, Y, Q),
    checkAll (X, L, Q).

--  check1(X,Y,Q)
--      Unify X,Y and succeed exactly once iff Q succeeds.
--      Do not bind X,Y, or Q however.
check1 (X, Y, Q) :-
    not (check2 (X, Y, Q)),
    !.
check1 (X, Y, Q) :-
    fail.

--  unify X,Y and see whether Q succeeds.  Fail is Q
--      succeeds & succeed if Q fails.
check2 (X, X, Q) :-
    Q,
    !,
    fail.
check2 (X, Y, Q).

presupposeNotEmpty ([]) :-
    write ('>>>>> Your input makes an invalid assumption.'),
    nl, !.
presupposeNotEmpty (X).

presupposeSingleton ([X]) :- !.
presupposeSingleton (L) :-
    write ('>>>>> Your input makes an invalid assumption.'), nl.
```

```
/************  OUTERMOST (PRINTING) OPERATORS  *****************/

yesNo (P) :-
    P,
    write ('>>>>> Yes.'), nl, !.
yesNo (P) :-
    write ('>>>>> No.'), nl.

declare (P) :-
    P,
    write ('>>>>> That is correct.'), nl, !.
declare (P) :-
    write ('>>>>> That is not correct.'), nl.

howMany (X, P, Q) :-
    solutions (X, P, L),
    presupposeNotEmpty (L),
    sumSolutions (X, L, Q, N),
    write ('>>>>> '), write (N), write ('.'), nl.

whoOrWhat (X, P, Q) :-
    solutions (X, P, L),
    presupposeNotEmpty (L),
    weedSolutions (X, L, M, Q),
    write ('>>>>> '), writeList (M).

listWhoOrWhat (X, nonQuestion(Y,P1,P2), Q) :-
        !, solutions (X, and(and(P1,P2),Q), L),
        write ('>>>>> '), writeList (L).

listWhoOrWhat (X, P, Q) :-
    solutions (X, P, L),
    presupposeNotEmpty (L),
    weedSolutions (X, L, M, Q),
    write ('>>>>> '), writeList (M).

which (plural, X, P, Q) :- !,
    solutions (X, P, L),
    presupposeNotEmpty (L),
    weedSolutions (X, L, M, Q),
    write ('>>>>> '), writeList (M).

which (singular, X, P, Q) :-
    solutions (X, P, L),
    weedSolutions (X, L, M, Q),
    presupposeSingleton (M),
    write ('>>>>> '), writeList (M).

sumSolutions (X, [], Q, 0).
```

```
sumSolutions (X, [Y|L], Q, N) :-
    check1 (X, Y, Q),
    !,
    sumSolutions (X, L, Q, M),
    plus (M, 1, N).
sumSolutions (X, [Y|L], Q, N) :-
    sumSolutions (X, L, Q, N).

writeList ([]) :-
    write ("None."), nl.
writeList ([X]) :-
    write (X),
    write ("."), nl.
writeList ([X,Y|L]) :-
    write (X),
    write (", "),
    writeList ([Y|L]).

weedSolutions (X, [], [], Q).
weedSolutions (X, [Y|L1], [Y|L2], Q) :-
    check1 (X, Y, Q),
    !,
    weedSolutions (X, L1, L2, Q).
weedSolutions (X, [Y|L1], L2, Q) :-
    weedSolutions (X, L1, L2, Q).
```

Example Queries and Responses

Next, several example queries along with the responses of the grammar are listed. Except for font changes and line breaking, the expressions in **boldface** were executed by the interpreter exactly as shown here, and the text in Courier font was the response produced by the grammar execution.

prog parseInput: 'Who taught cs 101?'.

```
TRANSLATION:
which(N; X:faculty; nameR(id=>X); defSingular(X2:cs101;
courseR (number=>lex101; dept=>cs; course=>X2; subject=>lexCS;
name=>[ lexCOMPILER, lexDESIGN ]); offeringR(year=>year;
course=>X2; faculty=>X)))

>>>>> dickKieburtz.
```

prog parseInput: 'Who has taught cs 101?'.

```
TRANSLATION:
which(singular; X:faculty; nameR(id=>X); defSingular(X2:cs101;
courseR(number=>lex101; dept=>cs; course=>X2; subject=>lexCS;
name=>[ lexCOMPILER, lexDESIGN ]); offeringR(year=>year;
course=>X2; faculty=>X)))

>>>>> dickKieburtz.
```

prog parseInput: 'Who is the professor who taught cs 101?'

```
TRANSLATION:
listWhoOrWhat(X:faculty; defSingular(X; and(P1:nameR(id=>X);
defSingular(X; P1; defSingular(X2:cs101; courseR(number=>lex101;
dept=>cs; course=>X2; subject=>lexCS; name=>[ lexCOMPILER,
lexDESIGN ]); offeringR(year=>year; course=>X2; faculty=>X))));
true); true)

>>>>> dickKieburtz.
```

prog parseInput: 'Did Kieburtz teach cs 101?'

```
TRANSLATION:
yesNo(defSingular(X:dickKieburtz; nameR(id=>X);
defSingular(X2:cs101; courseR(number=>lex101; dept=>cs;
course=>X2; subject=>lexCS; name=>[ lexCOMPILER, lexDESIGN ]);
offeringR(year=>year; course=>X2; faculty=>X))))

>>>>> Yes.
```

prog parseInput: 'Did Prof Kieburtz teach cs 102?'

```
TRANSLATION:
yesNo(defSingular(X:dickKieburtz; nameR(id=>X);
defSingular(X2:cs102; courseR(number=>lex102; dept=>cs;
course=>X2; subject=>lexCS; name=>[ lexARTIFICIAL,
lexINTELLIGENCE ]); offeringR(year=>year; course=>X2;
faculty=>X))))

>>>>> No.
```

prog parseInput: 'Kieburtz is teaching cs 101.'

```
TRANSLATION:
declare(defSingular(X:dickKieburtz; nameR(id=>X);
defSingular(X2:cs101; courseR(number=>lex101; dept=>cs;
course=>X2; subject=>lexCS; name=>[ lexCOMPILER, lexDESIGN ]);
offeringR(year=>year; course=>X2; faculty=>X))))
```

```
>>>>> That is correct.
```

prog parseInput: 'Prof Maier is teaching cs 101.'

```
TRANSLATION:
declare(defSingular(X:davidMaier; nameR(id=>X);
defSingular(X2:cs101; courseR(number=>lex101; dept=>cs;
course=>X2; subject=>lexCS; name=>[ lexCOMPILER, lexDESIGN ]);
offeringR(year=>year; course=>X2; faculty=>X))))

>>>>> That is not correct.
```

prog parseInput: 'Which courses did David Maier teach?'

```
TRANSLATION:
which(plural; Y:course; courseR(dept=>dept; course=>Y);
defSingular(X:davidMaier; nameR(id=>X); offeringR(year=>year;
course=>Y; faculty=>X)))

>>>>> cs100, cs102.
```

prog parseInput: 'Which course did Maier teach?

```
TRANSLATION:
which(singular; Y:course; courseR(dept=>dept; course=>Y);
defSingular(X:davidMaier; nameR(id=>X); offeringR(year=>year;
course=>Y; faculty=>X)))

>>>>> Your input makes an invalid assumption.
>>>>> cs100, cs102.
```

prog parseInput: 'What are the courses that Maier teaches?'

```
TRANSLATION:
listWhoOrWhat(X:course; defPlural(X; and(P1:courseR(dept=>D:dept;
course=>X); defSingular(X2:davidMaier; nameR(id=>X2);
defPlural(X; P1; offeringR(year=>year; course=>X;
faculty=>X2)))); true); true)

>>>>> cs100, cs102.
```

Hereafter, the responses produced by the execution of the grammar are omitted for brevity.

prog parseInput: 'The professor who taught cs 100 is teaching cs 102.'

```
prog parseInputNP: 'the courses that Maier teaches'.
prog parseInputNP: 'the course that Maier taught'.
prog parseInputNP: 'the course which Maier taught'.
prog parseInputNP: 'which courses in computer science'.
prog parseInputNP: 'every grade of Harry Porter in cs 101'.
prog parseInputNP: 'the student that got a B from Kieburtz in cs 101'.
prog parseInputNP: 'the student who had Kieburtz for cs 101'.
```

Biographical Note

The author entered this reality in the early hours of the morning, 14 April 1956, in Fort Worth, Texas. He graduated from Fort Vancouver High School in Vancouver, Washington, in 1974 and was promptly shipped to Brown University in Providence, Rhode Island, where he was issued a Bachelor of Science (Sc.B.) degree in Computer Science 4 years later.

He subsequently returned to the Portland, Oregon, area and worked for Applied Information Sciences, a small data processing company. After leaving this position, he founded a small business, Porter Systems, which contracted to establish a data processing department for its primary client, a clerical administration company. For them, he designed and implemented an online data entry and retrieval system for the processing of health and welfare claim forms, as well as a number of related H&W and pension administration systems. After successful completion of the project, Porter Systems was dissolved and the author began study at the Oregon Graduate Center.

The author married the most wonderful woman in the universe, Nancy McCarter, on 21 June 1986. To them was born a happy and healthy son, Graham Austin Porter, on 2 July 1987.

In addition to the topics of this dissertation, logic programming and computational linguistics, the author has broad interests in Computer Science, including the object-oriented paradigm, programming language environments, distributed information systems, neural modeling, and connectionist architectures.