

1986

# Expert system software tools :

Fabio J. Urbina  
*Lehigh University*

Follow this and additional works at: <https://preserve.lehigh.edu/etd>



Part of the [Electrical and Computer Engineering Commons](#)

---

## Recommended Citation

Urbina, Fabio J., "Expert system software tools : " (1986). *Theses and Dissertations*. 4705.  
<https://preserve.lehigh.edu/etd/4705>

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact [preserve@lehigh.edu](mailto:preserve@lehigh.edu).

# Expert System Software Tools OPS5 vs. Prolog

by

Fabio J. Urbina

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1986

This thesis is accepted and approved in partial fulfillment of the requirements for the Degree of Master of Science.

September 8, 1986  
(date)

Gerhard Rayna  
Professor in Charge

Donna J. Hillman  
CS Division Chairman

Donald L. Jolley, Jr.  
CSEE Department Chairman

## Acknowledgments

I would like to take this time to acknowledge all those who have helped to bring about this paper and the various concepts that are expressed.

Initially, I would like to thank my parents for their years of support, guidance, love and understanding without which I could not have produced this paper. I would also like to thank Dr. Gerhard Rayna for his guidance, and finally, I would like to thank all my friends and fellow graduate students who supported me throughout my graduate career.

# Table of Contents

## Abstract

1	Introduction	1
2	Representing Facts	1
3	Syntax	2
4	The Knowledge Base	2
4.1	OPS5 Uses Forward Chaining	3
4.2	Prolog uses Backward Chaining	4
5	The Inference Engine	5
5.1	The Prolog Inference Engine	5
5.2	OPS5 Execution Cycle	9
5.2.1	OPS5 Top Level Commands	10
5.2.2	Conflict Resolution Logic	11
6	Adding and Deleting Facts	12
7	OPS5 Compound Data Types	13
8	Simulating a Prolog example in OPS5	16
9	An Example of an Expert Grocery Bagging System	21
9.1	The Problem Description	21
9.2	The System Rules	23
9.3	OPS5 Implementation of BAGGER	26
9.3.1	The Data Items	26
9.3.2	Defining the Element Classes	27
9.3.3	Controlling the Flow of Execution	28
9.3.4	The OPS5 Production Rules for the Bagging System	29
9.3.5	Execution of the OPS5 BAGGER	35
9.4	Prolog Implementation of BAGGER	35
9.4.1	The Prolog Data Base	35
9.4.2	Controlling the Flow of Execution	36
9.4.3	The Prolog Production Rules for the Bagging System	37
9.4.4	Execution of the Prolog BAGGER	42
10	An Additional Example	42
10.1	The Problem Description	44
10.2	OPS5 Implementation of the Number Codes Example	46
10.3	Prolog Implementation of the Number Codes Example	51
11	Debugging Tools	54
11.1	Prolog Debugging Tools	54
11.1.1	The trace and notrace predicates	55
11.1.2	The spy and nospy predicates	55
11.1.3	The debugging predicate	56
11.2	OPS5 Debugging Tools	56
11.2.1	The run command	56
11.2.2	The watch command	56
11.2.3	The cs command	57
11.2.4	The pbreak command	57
11.2.5	The back command	57

12 Concluding Remarks	57
References	60
Vita	61

## List of Figures

<b>Figure 1:</b>	A list of grocery items and their properties	27
<b>Figure 2:</b>	Sample list of input for the OPS5 program	35
<b>Figure 3:</b>	Output of the OPS5 implementation of BAGGER	36
<b>Figure 4:</b>	Prolog version of a list of grocery items and their properties	36
<b>Figure 5:</b>	Sample list of input for the Prolog program	43
<b>Figure 6:</b>	Output of the Prolog implementation of BAGGER	43
<b>Figure 7:</b>	Binary tree depicting acceptable codes.	44
<b>Figure 8:</b>	Tree With Additional Nodes	46

## Abstract

This paper describes in some detail the Prolog and OPS5 languages. The main purpose is to point out the similarities as well as the differences between the two languages. The knowledge representation characteristics as well as the inference engines of each language are discussed. Finally, two examples and their implementation in each language are discussed to demonstrate their capabilities for problem solving.



## 1 Introduction

The programming language Prolog was invented by Alain Colmerauer at the University of Marseilles in the early 1970s. It was a first attempt at the design of a language that would enable a programmer to specify his tasks in logic, rather than in terms of conventional programming constructs about what the machine should do when. This explains how the name for the language was chosen. Prolog stands for PROgramming in LOGic [Clocksin & Mellish 84].

Even though Prolog is considered to be a general purpose language, it has been widely used in Europe for the implementation of expert systems, especially in the area of pharmacology [Brownston, et al. 85].

OPS5 is one of the latest versions in a series of languages developed at Carnegie-Mellon University for the modeling of human cognition and memory. Its name is an acronym for Official Production System, Version 5. OPS5 is a general purpose knowledge engineering language<sup>1</sup>, which has been widely used in the area of expert systems. Expert systems implemented in OPS5 include YES/MVS, MUD, and XCON [Waterman 86].

## 2 Representing Facts

The methods of representing facts in Prolog and OPS5 are essentially similar. The difference is that Prolog uses the predicate logic notation, while OPS5 uses the common LISP list notation. The table below is an example of how some simple facts are represented in both Prolog and OPS5.

Snoopy is a dog.  
Bill owns a car.  
The professor teaches math to the student.

---

<sup>1</sup>A knowledge engineering language is a programming language specifically designed for the development of expert systems.

## Prolog

```
dog(snoopy).  
owns(bill, car).  
teaches(prof, math, student).
```

## OPS5

```
(dog snoopy)  
(owns car bill)  
(teaches prof math student)
```

Note that all statements in Prolog must end with a period.

### 3 Syntax

The smallest units used for constructing Prolog and OPS5 programs are called terms. A term can be either a constant or a variable, and a constant can be either an atom or an integer. Any one of the following is considered to be an atom:

- a string of letters and digits that must begin with a letter (this letter must be lower-case in Prolog). The underscore character can be inserted inside the atom to improve legibility.
- a literal string. In Prolog, it must be enclosed in single quotes, in which case the atom can begin with an upper-case letter. In OPS5, strings are enclosed in a pair of bar symbols.

Variables are any combination of letters and digits, which may include the underscore character. In Prolog, a variable name must begin with either an upper-case letter or the underscore character. OPS5 on the other hand, does not distinguish between upper and lower-case letters, so variables must be enclosed in angle brackets.

### 4 The Knowledge Base

The knowledge base of an expert system contains the domain specific information necessary to solve a particular problem. This information is represented in the form of IF...THEN rules, such as the one below:

```
IF      the bug has no antennae, AND  
        the bug has eight legs,  
THEN   the class of the bug is arachnid.
```

Each rule is made up of a condition part, or premise, preceded by the word IF, and a conclusion, or action part preceded by the word THEN.

The OPS5 knowledge base consists of a set of facts and a set of rules called production rules which act on these facts. The facts are stored in an area of memory referred to as working memory, while the production rules are loaded into production memory. Production memory resembles a read only memory. Production rules can only be read and executed. They cannot be changed during execution. Working memory, on the other hand, can be changed constantly. This is one of the basic and essential features of OPS5. Prolog, on the other hand, does not make any distinction between working and production memory. All the facts and rules are stored in one global data base. It is possible, however, to add and delete facts to the data base, as in the OPS5 case.

Our main concern is how to represent rules in both Prolog and OPS5. This is where a major difference between the two languages stands out, and that is the fact that Prolog is inherently a backward chaining language, whereas OPS5 is a forward chaining one.

#### **4.1 OPS5 Uses Forward Chaining**

The forward chaining process moves from the condition parts to the action parts. If all the conditions in the premise are satisfied, the rule is said to be triggered. When the action part of a triggered rule is executed, the rule is said to fire. All the rules which are triggered, form a set from which only one rule will be chosen to fire. The process of selecting which particular rule from the set will fire is discussed in section 5.2.2.

In the example above, if the system determines that the bug has eight legs

and no antennae, and the arachnid rule is chosen to fire, then the system concludes that the class of the bug in question is arachnid.

In OPS5 the production rules are defined by a list whose first element is the constant symbol "p". The second term is a unique rule name, followed by lists of conditions and lists of actions. The lists of conditions are separated from the lists of actions by the symbol "-->". The name of a rule in OPS5 is for identification purposes only. This is because rules in OPS5 are not called by other rules or routines, as is the case with other computer languages including Prolog.

The arachnid rule in OPS5 would look like this:

```
(p class_arachnid
  (antennae 0)
  (legs 8)
-->
  (make class arachnid)
  (write |The class of bug is arachnid.|))
```

#### 4.2 Prolog uses Backward Chaining

The backward chaining process searches the data base through the conclusions of the rules. The user states a goal, and then Prolog attempts to satisfy the conditions which produce that goal. In the example rule above, the goal is to prove that the class of a particular bug is arachnid. To satisfy this goal, we must first satisfy two subgoals. The first subgoal states that the bug has no antennae, and the second subgoal states that the bug has eight legs. If these two subgoals can be satisfied, then the main goal will also be satisfied, and we can conclude that the class of the bug is indeed arachnid.

Rules in Prolog are written "backwards", that is, the action part is written before the condition part. A rule in Prolog is defined in the following

way:

$A :- B_1, B_2, \dots, B_n.$

The symbol “:-” is read IF, and the commas are read as AND. Therefore, the rule states that A is true if  $B_1, B_2, \dots, B_n$  are all true. Translated into Prolog, the arachnid rule from above would look like this:

```
class(arachnid) :- antennae(0),  
                  legs(8),  
                  print('The class of bug is arachnid.').
```

## 5 The Inference Engine

The inference engine uses the information in the knowledge base to select and execute the appropriate production rules. If a system is being implemented in a general purpose language, a considerable amount of time must be dedicated to the design of the inference engine. The advantage of using tools such as Prolog and OPS5 is that the inference engine is already built in.

### 5.1 The Prolog Inference Engine

Prolog is a language which is used to answer questions based on the facts stored in its data base. Given a question, Prolog will search for a fact or a rule in the data base which matches the question that was asked. A match exists if the predicates are the same, and if the number of arguments is also the same. When Prolog attempts to match a question being asked to a fact in the data base, we say that it is attempting to satisfy a goal.

Since Prolog, like most conventional programming languages, is sequential in its execution, the search is conducted in a top to bottom fashion. Suppose we have the following clauses in the data base.

```
mother(mary, tom).  
mother(mary, dick).  
mother(mary, sara).  
mother(jane, harry).
```

and to Prolog's prompt, we ask if Mary is the mother of Dick (user input is in boldface type):

```
?- mother(mary, dick).      Ask question.  
  
Yes.                        Prolog answers yes, and  
More? (Y/N):Y              asks if we want more. We respond yes.  
  
No.                          No more matches.
```

Prolog begins the search by attempting to match the clause at the top of the data base. Since the child's names are not the same, this attempt will fail, and Prolog proceeds to the next instance of the **mother** clause. At this point, Prolog will come back with a "yes", since it found a clause in the data base which matched the question exactly. If Prolog is not able to find a matching clause, it comes back with a "no". This, however, does not mean that the answer to the question is no. It simply means Prolog was not able to satisfy the given goal.

In addition, we can use a variable to ask Prolog to name all of Mary's children. The second argument is replaced with a variable name, say **Child**, and the question now looks like this:

```
?- mother(mary, Child).
```

The variable is initially uninstantiated, which means that it has no value. Prolog will then attempt to match the question to any clause in the data base which has **mother** as a predicate, and two arguments, the first of which must be **mary**. The second one can take any value, since a variable matches anything. When Prolog finds a match, the variable is said to be instantiated

to the value of the respective term in the matching data base fact. In response to the above question, Prolog will come back with:

```
Child = tom
More? (Y/N):
```

The variable **Child** has been instantiated to **tom**, because Prolog matched the question to the first instance of the **mother** clause. As we said before, Prolog is sequential and it conducts its search in a top to bottom fashion. Next, Prolog comes back and asks if we want to see Mary's other children. This continues until no match is found, at which point Prolog comes back with a "no" answer. The entire run would look like this:

```
?-mother(mary, Child).
```

```
Child = tom
More? (Y/N):Y
```

```
Child = dick
More? (Y/N):Y
```

```
Child = sara
More? (Y/N):Y
```

```
No.
```

Prolog can also handle satisfying several simultaneous goals. Suppose that the facts below were added to the data base.

```
father(bill, tom).
father(bill, sara).
father(bill, harry).
plays(tom, tennis).
plays(sara, soccer).
plays(harry, soccer).
```

We now wish to know which of the children of Mary and Bill play soccer. In other words, we want to find a person who is a child of Mary, and a child of Bill, and also plays soccer. The question to ask would be the following:

```
?- mother(mary, X), father(bill, X), plays(X, soccer).
```

Prolog attempts to satisfy the goals in a conjunction in a left to right order. Every time Prolog satisfies a goal, it marks the clause that satisfied it, then proceeds to the next goal on the right, and attempts to satisfy it. If this goal fails, it causes Prolog to backtrack to the previous goal. Prolog will then try to re-satisfy this goal beginning with the clause after the one that was marked, unless there isn't one, in which case the goal fails. If the goal is satisfied, Prolog will proceed to the right, otherwise, it will backtrack to the left. If there are no goals to the left, the search will end in failure.

Therefore, in the question above, when the first goal is satisfied, **X** is instantiated to **tom**, and Prolog moves to the right. The second goal is satisfied by the clause  
`father(bill, tom).`

The third goal, however, fails because there is no clause indicating that Tom plays soccer. This causes Prolog to backtrack and try to re-satisfy the second goal. This goal fails too, because there are no more clauses that have Bill as the father, and Tom as the child, so the first goal must be re-satisfied.

The next instance of **mother** causes **X** to be instantiated to **dick**. The search then continues to the right, but it does not go past the second goal because there is no **father** clause that has **dick** as the second argument (Dick's father is unknown).

Once again, the search starts over, moving downward in the **mother** clauses. **X** is now instantiated to **sara**, and this time, all the goals succeed. Prolog comes back and asks if we want to see more. If we respond yes, it will not even go past the first goal, since there are no more **mother** clauses with **mary** as the first argument. The run looks like this:



?- mother(mary, X), father(bill, X), plays(X, soccer).

X = sara  
More (Y/N):Y

No.

In the previous section we looked at the semantics of a Prolog rule. Prolog goes about trying to satisfy a particular rule in much the same way that it attempts to satisfy questions. The advantage of writing rules is that it allows large and complicated goals to be broken down into smaller, less complicated sub-goals. The reasoning is similar to that of writing sub-routines in languages such as Pascal and Fortran.

It should be clear by now that the search in Prolog is conducted in an exhaustive depth-first fashion.

## 5.2 OPS5 Execution Cycle

In OPS5, a rule can fire only if each one of its condition clauses succeeds. For a clause to succeed, there has to be an element in working memory with an equal number of terms, and each of the corresponding terms must match. For example, for the arachnid rule to fire, the following facts have to be in working memory:

(antennae 0)  
(legs 8)

Therefore, the facts which reside in working memory at any point in time are what determines which particular rule will fire. This is why OPS5 is said to be a data driven language. The sequence of events is controlled by the data, not by the order in which the rules are written.

When OPS5 is loaded into memory, it responds with the “->” prompt. This is the point where commands can be entered for OPS5 to interpret. The

production rules and the initial working memory elements are then loaded, and the **run** command is used begin the execution cycle. This cycle, also known as the recognize-act cycle, consists of two phases. The first phase is the "recognize" phase, and the second one is the "act" phase.

In the "recognize" phase, OPS5 compares the contents of working memory with the condition parts of the production rules. It then constructs the set of all the rules that can fire. This set is referred to as the conflict set, and the rules it contains are called instantiations in this particular context (which differs from the definition given in the Prolog context).

In the "act" phase, OPS5 chooses one rule from the conflict set according to some previously established criteria. The chosen rule will be the one to actually fire, and its action part is then executed. At this point the cycle starts over, and continues until it meets either of the following conditions:

1. A "(halt)" clause is encountered in the action part of the production rule being executed.
2. None of the condition parts of the rules in production memory is completely satisfied by the elements in working memory, in other words, the conflict set is empty.
3. An error is encountered, in which case execution is aborted.

### 5.2.1 OPS5 Top Level Commands

Commands in OPS5 have the same list format as facts. That is, they must be enclosed in parentheses. They can be entered at the top level, after the "-->" prompt. For example, to display the list of words in the dictionary, the **dict** command is used. It is entered in the following way:

```
-> (dict)
```

No period is necessary at the end of a line.

In OPS5 each of the facts residing in working memory is represented as a list, and there is a time tag associated with each list. This tag is an integer used to keep track of the order in which the facts were entered into working memory. Once a tag is assigned to a certain fact, it remains constant, so deleting a fact from working memory does not alter the time tag of any other fact.

### 5.2.2 Conflict Resolution Logic

To determine which rule from the conflict set will be the one to fire, OPS5 uses one of two conflict resolution strategies, namely LEX and MEA. There is only a slight difference between the two. The set of rules used in the LEX strategy is described below:

1. Refraction: to avoid loops, no rule is chosen to fire two consecutive times. That is, if after rule A has fired, both rule A and rule B can fire, then rule B will take precedence over rule A.
2. Recency: the rules are ordered by comparing the time tags of the working memory elements associated with the condition part of each rule, using the criteria that the most recent time tag has the highest value. If the first elements of each rule have the same tag, then the next elements are compared until either one rule comes out ahead, or the elements are exhausted. The chosen rule will then be the one with the highest value, or the one whose elements have not been exhausted.
3. Specificity: if no selection can be made by the previous rule, then other factors are taken into consideration, for example the number of condition clauses in the rules. The rule with the most condition clauses is then chosen, the reasoning behind this being that the greater the number of conditions, the more specific the rule is.
4. Finally, if all else fails, an arbitrary selection is made.

In the MEA strategy a rule is inserted in the above rule set, after the Refraction rule. This new rule compares the time tags of the working memory elements associated with the first condition of every rule in the conflict set, and

selects the one with the highest (most recent) time tag. This strategy is supposed to simulate the Means-Ends Analysis method of problem solving in the sense that it makes the orderly handling of subgoals easier. The first condition of the rules can then be set to be a goal, since the conflict resolution logic will always select a working memory element which matches a goal condition, instead of a more recent working memory element which does not match a goal condition.

## 6 Adding and Deleting Facts

In OPS5 facts are added to working memory by way of the **make** command. For example, typing the following line into the system

**(make legs spider 8)**

would enter the fact that spiders have eight legs into working memory with a time tag of 1. The next element entered would have a time tag of 2, and so on. The higher the value of the time tag, the more recent the element is considered to be. It is possible to enter the same fact into working memory more than once. This may be useful and deliberate in some cases, but usually it is a source, or consequence, of error.

To view the contents of working memory, the **wm** command is used. To delete an element from working memory, we use the **remove** command, along with the time tag of the fact to be deleted. For example,

**(remove n)**

deletes the element which has a time tag of  $n$  from working memory. This command can also be used in the action part of a production rule, in which case the  $n$  refers to the working memory element which matched the  $n^{\text{th}}$  clause

in the condition part of the rule.

Similarly, it is possible to add and delete facts to the Prolog data base by using several built-in predicates. To add facts to the data base, the **asserta** or **assertz** predicates can be used. The difference is that **asserta** adds the fact to the top, or beginning, of the data base, whereas **assertz** adds it to the bottom, or end, of the data base. For example,

```
asserta(mother(mary, tom)).
```

adds the fact that Mary is the mother of Tom to the top of the data base.

To remove facts from the Prolog data base, the **retract** predicate is used. This predicate, if it succeeds, will retract any fact which matches its argument. For example, by typing a "Y" at the "More" prompt, we can delete all of Mary's children from the data base as follows:

```
?- retract(mother(mary, Child)).
```

```
Child = tom  
More? (Y/N):Y
```

```
Child = dick  
More? (Y/N):Y
```

```
Child = sara  
More? (Y/N):Y
```

```
No.
```

## 7 OPS5 Compound Data Types

OPS5 programs, in general, contain a declaration section, where compound data structures can be declared. The compound data structure definable in OPS5 is called an element class. It is a list preceded by the keyword **literalize**, and it is composed of a class name, followed by any number of attributes. The general form is:

```
(literalize class-name attribute1
                        attribute2
                        .
                        .
                        attributen )
```

The number of attributes allowed in an element class depends on the particular implementation of OPS5. All attributes are initially set to the value "nil". This means that nothing is known about the value of the attributes. An example of an element class is:

```
(literalize Person      ; Element class representing a person
  name                  ; proper name of the person
  mother                ; name of the person's mother
  father               ; name of the person's father
  age                  ; age of the person
)
```

To reference attributes in an element class, the prefix operator "^" is used in front of the attribute name. If a value follows, it is taken to be the attribute value. Otherwise, the attribute value becomes "nil", the default. With the use of the "^", the attributes can be referenced by name regardless of order. For example, the condition

```
(Person ^name sara ^father bill ^mother mary)
```

would match the working memory element

```
* (Person ^name sara ^mother mary ^father bill)
```

Even if one of the attributes is absent in the condition, the working memory element would still match. For example, the condition

```
(Person ^name sara ^father bill)
```

would also match the above working memory element.

If the element class contains only one attribute, it is not necessary to specify the attribute name. For example, if the above element class contained

only the **name** attribute of the person, then the condition

```
(Person sara)
```

would match the working memory element

```
(Person ^name sara)
```

It is not necessary to specify the attribute name in working memory either, but it is done in this case for demonstration purposes. Note also that it is not necessary to capitalize element class names since, as we stated before, most implementations of OPS5 do not differentiate between upper and lower case.

There is a special type of attribute, called a vector-attribute, which can be included in an element class. The only restriction is that each element class can contain at most one vector-attribute. The function of a vector-attribute is to allow a particular attribute to have more than one value. As is the case with an element class, a vector-attribute must also be defined in the declaration section. The general form is:

```
(vector-attribute attribute1
                    attribute2
                    .
                    .
                    attributen)
```

For example, suppose we have the following element class declaration:

```
(literalize Room name contents)
```

If we want the attribute **contents** to represent a list of the contents of the room, we can declare it as a vector-attribute, as follows:

```
(vector-attribute contents)
```

We can then use the **make** command to define the bedroom and its contents:

```
(make Room
  ^name bed_room
  ^contents bed chair table lamp dresser window)
```

## 8 Simulating a Prolog example in OPS5

Unlike Prolog, OPS5 does not have a built in question answering ability, and therefore it must be simulated. Consider the example which was used in section 5.1. The `wm` command is used to view the contents of working memory:

```
-> (wm)
    4 (mother jane harry)
    3 (mother mary sara)
    2 (mother mary dick)
    1 (mother mary tom)
```

The numbers on the left are the time tag of each element.

A production rule is necessary to list all of Mary's children. The first attempt to write such a rule is the following:

```
(p mother_mary
  (mother mary <child> )
-->
  (write |Mary is the mother of| )
  (write <child> )
  (write (crlf) ) )
```

Notice the use of variables in OPS5. In the rule above, the condition clause will match any three term element in working memory whose first and second terms are `mother` and `mary` respectively. A variable matches any term, but this causes the variable to become bound to the value of the term.

Suppose the above rule is loaded, and the `run` command is entered to begin execution. The condition clause

```
(mother mary <child>)
```

matches the working memory element with the highest time tag, which in this



case is 3. The variable <child>, hence, becomes bound to **sara**. The rule fires and prints out:

**Mary is the mother of SARA**

The execution cycle continues since there remain elements in working memory which match the condition clause. The next time, the element with the second highest time tag is chosen, and <child> gets the value **dick**. The third time it would be expected that the element chosen would be the one with the next lowest time tag. This is not the case, however. Instead, the element with the highest tag is chosen once again. The system goes into an infinite loop. If the rule were allowed to fire four times, execution would look like this:

**Mary is the mother of SARA**  
**Mary is the mother of DICK**  
**Mary is the mother of SARA**  
**Mary is the mother of DICK**

OPS5 matches the rule condition to the working memory element with the highest, or most recent time tag. The next time, however, it seems to remember not to use the same element two times consecutively, so it selects the element with the next highest time tag. However, because it cannot remember more than one element, the third time it goes back and selects the first element once again. OPS5 continues, alternating between the first and second elements, as long as there are elements which match the condition. This is the reason for the infinite loop.

Therefore, to avoid running into infinite loops, working memory must be modified every time the rule fires. More specifically, the element which caused the rule to fire must be removed from working memory. Execution will now terminate when there are no more facts which match the condition clause. The rule must be modified as follows:

```

(p mother_mary
  (mother mary <child> )
-->
  (remove 1)
  (write |Mary is the mother of| )
  (write <child> )
  (write (crlf) )
)

```

Notice how the **remove** command is used to delete the element which matched the condition clause. Execution looks like this:

```

-> (run)
Mary is the mother of SARA
Mary is the mother of DICK
Mary is the mother of TOM

```

The loop has been eliminated, but there is a drawback, and that is that the facts have been deleted from the data base, and there is no way to recover them. To eliminate this problem, we can add a fourth attribute to each element that would specify initially, that the element has not been printed. Then after the element is printed, the attribute value can be modified to indicate that the action has taken place. The **modify** command can be used to change one or more attribute values in a working memory element. Obviously, the same task could be accomplished by removing the element from working memory, and then creating a new one with the appropriate changes, but **modify** provides two advantages. The first and most apparent, is that by this being a single built-in command, it is more efficient. The second is that when **modify** is used, the time tag of the working memory element which is being modified remains unaltered. This may be useful in some applications where preservation of time tags is imperative, for example, when using the MEA conflict resolution strategy.

With the addition of the fourth attribute, the elements in working memory

now look like this:

```
4 (mother jane harry not_printed)
3 (mother mary sara not_printed)
2 (mother mary dick not_printed)
1 (mother mary tom not_printed)
```

and the new rule looks like this:

```
(p mother_mary
  mother mary <child> not_printed )
-->
  (modify 1 ^4 printed )
  (write |Mary is the mother of| )
  (write <child> (crlf) )
)
```

The first condition clause must now check that the value of the fourth attribute is "not-printed." This means that the message which indicates that Mary is the mother of this particular child has not yet been printed.

The numbers in the first action clause translate to the following:

- The number 1 specifies that the working memory element to be modified is the one which matched the first condition clause in the production rule.
- The number 4 preceded by the caret character indicates that the value of the fourth attribute of the working memory element is the one to be modified. In this case, the value is changed to "printed."

The result of the execution of this rule is then:

```
-> (run)
Mary is the mother of SARA
Mary is the mother of DICK
Mary is the mother of TOM
```

In much the same way as above, a production rule can be written in OPS5 which simulates Prolog's ability to satisfy simultaneous goals. The result is the following rule:

```

(p sport
  (mother mary <child> )
  (father bill <child> )
  (plays <child> soccer)
-->
  (remove 3)
  (write <child> )
  (write |plays soccer.| )
  (write (crlf) )
)

```

To avoid loops, we choose the method of removing working memory elements. It is necessary to eliminate one, and only one, of the elements which caused the rule to fire. In this case the one which matches the third clause is chosen arbitrarily.

Suppose working memory contains the following facts:

```

10 (plays harry soccer)
9 (plays sara soccer)
8 (plays tom tennis)
7 (father bill harry)
6 (father bill sara)
5 (father bill tom)
4 (mother jane harry)
3 (mother mary tom)
2 (mother mary dick)
1 (mother mary sara)

```

When the rule is executed, OPS5 attempts to match the first condition to one of the elements in working memory. The one with the highest time tag is 3, and the value of <child> becomes **tom**. Once a variable in a particular rule has been bound to a certain value, it will retain that binding in all subsequent condition and action clauses of the rule. Therefore, OPS5 will try to satisfy the second condition as if it were

```
(father bill tom)
```

and will succeed.

However, since there is no element which matches the condition,

```
(plays tom soccer)
```

OPS5 starts over by attempting to find another match for the first condition. This is very similar to the way in which Prolog goes about attempting to satisfy goals. The difference is that Prolog marks a clause once it has matched, and "remembers" not to use it again. That is why it is not necessary to remove facts from the data base, as is the case with OPS5. OPS5 will keep firing a rule as long as there are elements in working memory which match its conditions, regardless of whether the results it produces are identical or not.

The result of the execution is then:

```
--> (run)
      SARA plays soccer.
```

## 9 An Example of an Expert Grocery Bagging System

In this section, we develop a rule-based system, which performs the task of solving a simple problem. The system is then implemented in both OPS5 and Prolog. The problem is based on a simplified version of the BAGGER example discussed in Chapter 6 of [Winston 84].

### 9.1 The Problem Description

The program is supposed to simulate a check-out clerk at a grocery store. The program will start the process by bagging large items, taking care to put large bottles in first. Every time a bag becomes full, the program restarts with a fresh bag. When it is done bagging the large items, the program will start a fresh bag, and begin bagging the small items. This process is similar to that of bagging large items except that frozen items must be placed in individual freezer bags. The process continues until there are no more items to be bagged.


In the original example, grocery items come in three different sizes, namely small, medium, and large. In this example we have restricted the sizes to be

only small or large. There are two reasons for this. The first reason is that the program keeps no record of what is in each bag. Small items in the original example are put in partially filled bags wherever there is room, but since our program is not able to determine where there is room, we make the small class of our example include the small and medium classes of the original example. The second reason is that, even if the program did keep track of what is in each bag, the number of combinations of small, medium, and large items which make up a full bag is relatively large, and it would be considerably difficult to determine when exactly a bag becomes full. Each bag, therefore, is considered to be full when it contains six, either large or small, items.

The grocery bagging task has been subdivided into three major steps. The first step merely initiates the bagging process. The second step bags the large items, and finally, the third step bags the small items. The original example also includes a check-order step which has been omitted here, although it would not be difficult to incorporate.

The example, as stated originally, is very well suited for implementation in OPS5 for two reasons. The first reason is that BAGGER is based on IF...THEN rules which support forward chaining. The second reason is that, from the rules, it builds a conflict set, and then uses conflict resolution to select a rule to fire.

There are two conflict resolution methods used by BAGGER. The first method is referred to as context limiting. With this method, the likelihood of conflict is reduced by separating the rules into groups. Only one group will be active at any point in time, and groups can be activated and deactivated. This can be done on OPS5, by setting goals, and it will be discussed in more detail



in section 9.3.3. The second method, however, is specificity, and cannot be properly implemented in OPS5. The reason is that, even though OPS5 uses specificity in conflict resolution, recency always takes priority, and this cannot be changed. It is therefore necessary to modify the original rules to overcome this minor difficulty.

## 9.2 The System Rules

The first step in the process commands the program to begin the bagging process. Context limiting is achieved by making the first condition clause of each rule limit the rule to the step being executed. As we stated before, the bagging process is divided into three steps, namely **begin-bagging**, **bag-large-items**, and **bag-small-items**. Initially, the step will be **begin-bagging**, and all the rules included in this step must check for it in their first condition clause.

There are three rules for the first step:

**begin-bagging-1**

```
If the step is begin-bagging
   there is a large item to be bagged
then discontinue the begin-bagging step
   start the bag-large-items step
```

**begin-bagging-2**

```
If the step is begin-bagging
   there are no large items to be bagged
   there is a small item to be bagged
then discontinue the begin-bagging step
   start the bag-small-items step
```

**begin-bagging-3**

```
If the step is begin-bagging
   there are no large items to be bagged
   there are no small items to be bagged
then discontinue the begin-bagging step
   halt execution
```

The first rule, **begin-bagging-1**, checks if there are any large items to be bagged, and if so, changes the step to **bag-large-items**. In other words, it

initiates the bag large items process. The second rule executes if there are no large items to be bagged, but there are unbagged small items. In this case, it will change the step to bag-small-items. Finally, begin-bagging-3 executes only if there are no items to be bagged, and terminates execution.

The second step, bag-large-items, consists of five rules. The first rule checks if there are any bottles among the large items to be bagged, and ensures that they are put in first. It also checks that each bag has less than six items before putting another item into a bag. The second rule checks that there are no more large bottles in the order, and if there are any remaining large items to be bagged, it bags them, again checking that no more than six items go into any one bag. The first two rules are listed below:

bag-large-items-1

```
If   the step is bag-large-items
     there is a large bottle to be bagged
     the bag contains < 6 large items
then put the large bottle in the bag
```

bag-large-items-2

```
If   the step is bag-large-items
     there are no large bottles to be bagged
     there is a large item to be bagged
     the bag contains < 6 large items
then put the large item in the bag
```

Rule bag-large-items-3 deals with starting a new bag after the current one has been filled. There is one consideration to take before opening a new bag, however. The system must check if there are more large items still unbagged. If there are, then a new bag can be started, and the process continues. Rule bag-large-items-3 is listed below:

bag-large-items-3

```
If   the step is bag-large-items
     there is a large item to be bagged
     the bag is full
then start a fresh bag
```

When there are no more large items remaining, but there are small items



to be bagged, rule **bag-large-items-4** takes over. It changes the step to **bag-small-items**, and starts a new bag regardless of whether or not the current bag was full. The reasoning for this is that large items are bagged in large bags, and small items are bagged in small bags. Rule **bag-large-items-4** looks like this:

```
bag-large-items-4
  If   the step is bag-large-items
       there are no large items to be bagged
       there is a small item to be bagged
  then discontinue the bag-large-items step
       start the bag-small-items step
       start a fresh bag
```

The last of this set of five rules executes when there are no more items to be bagged, either large or small. In this case execution is terminated. The rule is listed below:

```
bag-large-items-5
  If   the step is bag-large-items
       there are no large items to be bagged
       there are no small items to be bagged
  then discontinue the bag-large-items step
       halt execution
```

The **bag-small-items** step is very similar to the previous step. Items are bagged one by one in any order, starting new bags when necessary. The only special case is that of frozen items which must be enclosed in insulated freezer bags. When no more items remain unbagged, the process terminates. The rules are listed below:

```
bag-small-items-1
  If   the step is bag-small-items
       there is a small item to be bagged
       the item is not frozen
       the bag contains < 6 small items
  then put the small item in the bag
```

#### **bag-small-items-2**

```
If the step is bag-small-items
   there is a small item to be bagged
   the item is frozen
   the bag contains < 6 small items
then put the small item in an insulated freezer bag
   put the small item in the bag
```

#### **bag-small-items-3**

```
If the step is bag-small-items
   there is a small item to be bagged
   the bag is full
then start a fresh bag
```

#### **bag-small-items-4**

```
If the step is bag-small-items
   there are no items to be bagged
then discontinue the bag-small-items step
   halt execution
```

### **9.3 OPS5 Implementation of BAGGER**

#### **9.3.1 The Data Items**

The original bagger contains two sets of information. The first one is a list of all the items, with their respective properties, which are available in the supermarket. The second set is a list of the items included in a particular customer order. For the OPS5 program, the two sets have been combined. That is, every time an item is included in an order, it is listed with all its properties. This may seem to be wasteful and redundant. However, it does help to keep the production rules simple, especially when negated conditions have to be used. A list of all the available items and their properties is shown in figure 1.

In addition to the information about the items in the order, we also require information about the bags being filled. We made the decision not to keep a record of what specific items are in each bag. It is necessary, however, to keep track of the number of items in the bag which is being filled at any

<i>Item</i>	<i>Container</i>	<i>Size</i>	<i>Frozen</i>	<i>Status</i>
Bread	Plastic bag	Small	No	Unbagged
Peanut butter	Jar	Small	No	Bagged
Granola	Box	Large	No	Unbagged
Ice cream	Carton	Small	Yes	Unbagged
Soda	Bottle	Large	No	Bagged
Potato chips	Plastic bag	Small	No	Unbagged
TV dinner	Box	Small	Yes	Bagged

**Figure 1:** A list of grocery items and their properties

point in time. Each bag is referenced by an integer index which is incremented every time a fresh bag is started. With each bag, we also associate another integer which is used to count the number of items in the bag. This counter is incremented every time a new item is put in the bag, and reset to zero every time a fresh bag is started. The **bag** class then, contains the following attributes:

- **bag-index:** refers to the bag currently being filled
- **num-items:** refers to the number of grocery items which are in the bag currently being filled

### 9.3.2 Defining the Element Classes

We must define an element class to represent the physical objects that the problem deals with. The first element class defines grocery items as follows:

```

(literalize item      ; Element class to represent a grocery item
  name                ; name of the item
  container           ; type of container the item comes in
  size                ; either large or small
  frozen              ; either yes or no
  status              ; either bagged or unbagged
)

```

The second element class defines the bag being filled in the following way:

```

(literalize bags      ; Element class to represent the
                      ; bag being filled
  bag_index           ; index used to refer to the bag
  num_items           ; number of items in the bag
)

```

### 9.3.3 Controlling the Flow of Execution

The program to solve this problem uses a goal-driven strategy. As we stated before, the process of bagging groceries can be divided into three steps, which must be executed in a prespecified order. By setting goals, it is possible to control the flow of execution. The program starts with a goal to complete a particular step. When that step is completed, a new goal is set to complete the next step, and the process continues until the final state is reached. For this problem, the initial goal is to get the bagging process started. After this goal has been achieved, the next goal is to bag the large items, and finally, the last goal is to bag the small items.

Only one goal can be active at any point in time, and the first condition clause in every production rule test which step is active at that time. This ensures that when the step is **bag-large-items**, for example, only the rules which deal with bagging large items are able to fire.

To represent the goals, we must define yet another element class. The definition follows:

```

(literalize step      ; Element class to represent a step or goal
 step_name)          ; name of the step. Can be one of:
                    ; begin-bagging, bag-large-items, or
                    ; bag-small-items
)

```

### 9.3.4 The OPS5 Production Rules for the Bagging System

In this section, we attempt to draw the analogy between the rules discussed in the previous section, and the rules written specifically in OPS5.

Let us consider the first three rules to introduce some new concepts:

```

(p begin_bagging_1
 { (step begin_bagging) <step0> }
 (item ^size large ^status unbagged )
-->
 (remove <step0> )
 (make step ^step_name bag_large_items )
 (make bags ^bag_index 1 ^num_items 0 )
 (write (crlf) |Bag      1| )
)

(p begin_bagging_2
 { (step begin_bagging) <step0> }
 -(item ^size large ^status unbagged )
 (item ^size small ^status unbagged )
-->
 (remove <step0> )
 (make step ^step_name bag_small_items )
 (make bags ^bag_index 1 ^num_items 0 )
 (write (crlf) |Bag      1| )
)

(p begin_bagging_3
 { (step begin_bagging) <step0> }
 -(item ^size large ^status unbagged )
 -(item ^size small ^status unbagged )
-->
 (remove <step0> )
 (write (crlf) |There are no items to be bagged.| )
 (halt )
)

```

The previous rules are the ones used to initiate the bagging process. Note that the first condition clause of each rule tests for the step name to be **begin-bagging**. Note also that in the first condition clause, there is a variable named **step0**. This variable is an instance of what is referred to as an element

variable. An element variable, denoted like any other variable, is a label for the condition element which is associated with it. The association is made by enclosing the condition element and the element variable in curly brackets. Once an element variable has been associated with a condition element, it can be used in the action part of the rule to refer to the working memory element which matches the condition element. Working memory elements can thus be removed or modified.

A particular element variable may appear only once in the condition part of a rule. However, OPS5 permits an ordinary variable to have the same name as an element variable in the same rule. The scope of an element variable is a single rule.

The only requirement for this program to execute properly is that the element

```
(step begin_bagging )
```

be in working memory before the "run" command is entered, since all the rules capable of initiating the process test for this element. In all the rules, the element variable `step0` is associated with this working memory element.

Note that the element class `step` is used for reasons of formality more than anything else. We could have just as well replaced the clause

```
(remove <step0>)
```

with

```
(remove 1)
```

to remove the working memory element which matches the first condition, and we would not have had the need for an element variable. The first condition in the rule would have then simply been

`(step begin_bagging)`

This is also true for all subsequent rules.

The purpose of the first rule is to initiate the bag large items step, so the second condition clause checks that there are indeed large items in the customer order: in other words, that there is an element in working memory of class item, whose value for the attribute size is large, and whose value for the attribute status is unbagged. If the rule fires, the first action clause will delete the working memory element associated with `step0`. The second clause will then change the step name to be bag-large-items. It does this by creating a working memory element whose class is step, and whose value for the attribute step-name is bag-large-items. This action is intended to make possible the triggering of all the rules which deal with bagging large items, i.e. the following set of rules, in the next execution cycle. The next action simply creates an element which indicates that the first bag has been started, and that it contains no grocery items. Finally, the last clause prints a message which informs the user that the first bag is being filled with groceries.

The second rule introduces the concept of negation. Any condition element can be negated except for the first one. A negated condition succeeds if there is no element in working memory which matches the condition element. In this case, the second condition clause will succeed if there are no items of size large, whose status is unbagged.

The action part of the second rule is very similar to that of the first, with the only exception of the second action clause. In this case, since there are no large items to be bagged, the rule initiates the bag small items step. Note that the rule does check that there are small items to be bagged.

Finally, the third rule in this group fires when the step is begin-bagging, and there are no items to be bagged. It removes the step element from working memory, and prints out an appropriate message.

The rules for the second step, bag-large-items, are listed below:

```

bag_large_items_1
{ (step bag_large_items) <step1> }
{ (item ^name <name> ^size large ^container bottle
  ^status unbagged ) <groc_item> }
{ (bags ^num_items {<num> < 6}) <this_bag> }
-->
(modify <groc_item> ^status bagged )
(modify <this_bag> ^num_items (compute <num> + 1) )
(write (crlf) |      | <name> )
)

(p bag_large_items_2
{ (step bag_large_items) <step1> }
-(item ^size large ^container bottle ^status unbagged )
{ (item ^name <name> ^size large ^status unbagged ) <groc_item> }
{ (bags ^num_items {<num> < 6}) <this_bag> }
-->
(modify <groc_item> ^status bagged )
(modify <this_bag> ^num_items (compute <num> + 1) )
(write (crlf) |      | <name> )
)

(p bag_large_items_3
{ (step bag_large_items) <step1> }
(item ^size large ^status unbagged )
{ (bags ^bag_index <num1> ^num_items {<num2> >= 6}) <this_bag> }
-->
(modify <this_bag> ^bag_index (compute <num1> + 1) ^num_items 0 )
(write (crlf) (crlf) |Bag| (compute <num1> + 1) )
)

(p bag_large_items_4
{ (step bag_large_items) <step1> }
-(item ^size large ^status unbagged )
(item ^size small ^status unbagged )
{ (bags ^bag_index <num1>) <this_bag> } -->
(remove <step1> )
(make step bag_small_items )
(modify <this_bag> ^bag_index (compute <num1> + 1) ^num_items 0 )
(write (crlf) (crlf) |Bag| (compute <num1> + 1) )
)

```



```

(p bag_large_items_5
  { (step bag_large_items) <step1> }
  -(item ^size large ^status unbagged )
  -(item ^size small ^status unbagged )
-->
  (remove <step1> )
  (write (crlf) (crlf) |Bagging completed!! )
  (halt )
)

```

In the first action clause of the first rule, the status of the item is changed to **bagged** by means of the **modify** command, to indicate that the item has been put in the bag. The second action clause uses the **modify** command in combination with the **compute** command. The latter command is used to perform arithmetic computations in OPS5. In this case it is being used to replace the value of the attribute **num-items** with the result of the computation. This attribute represents the number of items in the bag currently being filled, and it is incremented by one to indicate that a new item has been placed in the bag. The **compute** command can also be used inside a **write** command, as is the case in the last action clause of the rule **bag-large-items-3**.

It is important to note that commands such as **make**, **remove**, **modify**, and **write** can be used in production rules, but only in the action part of the rules. The reason for this is that, if they were included in the condition part, they would not match any working memory elements, and would cause the rule to always fail.

The last group of rules is listed below. The rules are very similar in purpose and function to those of the first group.

```

(p bag_small_items_1
  { (step bag_small_items) <step2> }
  { (item ^name <name> ^size small ^status unbagged ) <groc_item> }
  { (bags ^num_items {<num> < 6}) <this_bag> }
-->
  (modify <groc_item> ^status bagged )
  (modify <this_bag> ^num_items (compute <num> + 1) )
  (write (crlf) |      | <name> )
)

(p bag_small_items_2
  { (step bag_small_items) <step2> }
  { (item ^name <name> ^size small ^frozen yes
      ^status unbagged ) <groc_item> }
  { (bags ^num_items {<num> < 6}) <this_bag> }
-->
  (modify <groc_item> ^status bagged )
  (modify <this_bag> ^num_items (compute <num> + 1) )
  (write (crlf) |      | <name> |in insulated freezer bag| )
)

(p bag_small_items_3
  { (step bag_small_items) <step2> }
  (item ^size small ^status unbagged )
  { (bags ^bag_index <num1> ^num_items {<num2> >= 6}) <this_bag> }
-->
  (modify <this_bag> ^bag_index (compute <num1> + 1) ^num_items 0 )
  (write (crlf) (crlf) |Bag| (compute <num1> + 1) )
)

(p bag_small_items_4
  { (step bag_small_items) <step2> }
  -(item ^status unbagged )
-->
  (remove <step2> )
  (write (crlf) (crlf) |Bagging completed!! )
  (halt )
)

```

Note that the rule `bag-small-items-1` could bag small frozen items, if the rule `bag-small-items-2` were not present. Whenever there is a frozen item to bag, the conflict set includes both rules. However, because of specificity, conflict resolution always selects `bag-small-items-2`. Even for a sample of input which contains only frozen items, `bag-small-items-1` will never fire.

### 9.3.5 Execution of the OPS5 BAGGER

Figure 3 shows the results of a run of the OPS5 version of BAGGER using the sample input shown in figure 2.

```
(step begin_bagging )  
  
(item ice_cream carton small yes unbagged )  
(item peanut_butter jar small no unbagged )  
(item soda bottle large no unbagged )  
(item tv_dinner box small yes unbagged )  
(item soda bottle large no unbagged )  
(item granola box large no unbagged )  
(item soda bottle large no unbagged )  
(item potato_chips plastic_bag small no unbagged )  
(item soda bottle large no unbagged )  
(item granola box large no unbagged )  
(item bread plastic_bag small no unbagged )  
(item soda bottle large no unbagged )  
(item ice_cream carton small yes unbagged )  
(item peanut_butter jar small no unbagged )  
(item granola box large no unbagged )
```

Figure 2: Sample list of input for the OPS5 program

## 9.4 Prolog Implementation of BAGGER

### 9.4.1 The Prolog Data Base

The Prolog data base, like the original example, contains two lists of information. There are two reasons for this. First, since the `bagged` clause contains only one attribute, it is easier to retract an item from the data base once it has been bagged, and second, using two different lists doesn't make the rules overly complicated. The Prolog version of the list of available items and their properties is shown in figure 4.

```

Bag      1
        SODA
        SODA
        SODA
        SODA
        SODA
        GRANOLA

Bag      2
        GRANOLA
        GRANOLA

Bag      3
        ICE_CREAM in insulated freezer bag
        PEANUT_BUTTER
        TV_DINNER in insulated freezer bag
        POTATO_CHIPS
        BREAD
        ICE_CREAM in insulated freezer bag

Bag      4
        PEANUT_BUTTER

Bagging completed!

```

Figure 3: Output of the OPS5 implementation of BAGGER

```

item(granola,      box,      large, no ).
item(peanut_butter, jar,    small, no ).
item(ice_cream,   carton,   small, yes).
item(soda,        bottle,   large, no ).
item(bread,       plastic_bag, small, no ).
item(potato_chips, plastic_bag, small, no ).
item(tv_dinner,   box,      small, yes).

```

Figure 4: Prolog version of a list of grocery items and their properties

#### 9.4.2 Controlling the Flow of Execution

The flow of execution in Prolog is controlled by a main rule which calls other rules in the appropriate order. In this program, it is essential that all the rules be executed, so it is necessary to prevent any rule in the program from failing. Otherwise, the entire program will fail. We must therefore make sure that there is no set of input which will cause any particular rule to fail. For

example, we must ensure that the rule `bag-large-items` succeeds every time, even if there are no large items to be bagged.

The main rule, `bag`, is shown below:

```
bag :- init_bag,  
       bag_large_items,  
       start_new_bag,  
       bag_small_items,  
       completed_bagging,  
       !.
```

The character `“!”` is a Prolog predicate referred to as the `“cut”`. As a goal, it always succeeds, and its purpose is to ensure that once it has been satisfied, Prolog will not attempt to resatisfy any of the rules to the left of the `“cut”`. For a more detailed explanation of the `“cut”`, see [Clocksin & Mellish 84].

The rules invoked by `bag` are discussed subsequently.

#### 9.4.3 The Prolog Production Rules for the Bagging System

The rule `init-bag` is used to initialize the bag index, and the item counter. Both are asserted as facts to the data base, so that they can be accessed by other rules. The first condition in `init-bag` makes use of what is known as the anonymous variable, represented by the underscore character. The anonymous variable, like all variables, will match anything, but it does not become instantiated. It is used when the variable will not be used elsewhere in the clause, and the programmer does not want to come up with a creative name. The first condition will, therefore succeed if there is at least one item to be bagged. If there are no unbagged items, the first clause will fail. Prolog will attempt to resatisfy the goal, and the second clause executes. This clause will always succeed. It asserts the fact that there were no items to be bagged in the data base, and it prints out an appropriate message. The rule is shown

below:

```
init_bag :- unbagged(_),
            asserta(bag_index(1)),
            asserta(num_items(0)),
            print('Bag 1'), nl.
init_bag :- asserta(no_items),
            print('There are no items to be bagged. '), nl.
```

The rule **bag-status** is called before an item is put in a bag. In the first condition of the first clause, the variable **N** will be instantiated to the number of items in the bag currently being filled. If that number is strictly less than six, the first clause will succeed. Otherwise, it will fail, and the second clause will execute. The bag index is incremented, and the number of items in the bag is reset to zero. Note that the second clause will never fail. The rule is shown below:

```
bag_status :- num_items(N),
              N < 6.
bag_status :- retract(bag_index(I)),
              retract(num_items(N)),
              J is I + 1,
              asserta(bag_index(J)),
              asserta(num_items(0)),
              nl, print('Bag ', J), nl.
```

The next step in the process is to bag the large items. As we already know, large bottles must be bagged before anything else. The rule **bag-large-items** takes care of bagging large items in the appropriate order. It is listed below:

```

bag_large_items :- unbagged(Name),
                  item(Name, bottle, large, _),
                  bag_status,
                  retract(num_items(N)),
                  M = N + 1,
                  asserta(num_items(M)),
                  retract(unbagged(Name)),
                  write("    ", Name), nl,
                  bag_large_items.
bag_large_items :- unbagged(Name),
                  item(Name, _, large, _),
                  bag_status,
                  retract(num_items(N)),
                  M = N + 1,
                  asserta(num_items(M)),
                  retract(unbagged(Name)),
                  write("    ", Name), nl,
                  bag_large_items.
bag_large_items.

```

Let us consider the first clause only. The first condition checks that there is indeed an item to be bagged, and the variable **Name** becomes instantiated to the name of that item. The second condition will succeed only if the item is a large bottle. Otherwise, Prolog will attempt to resatisfy the first condition, and the variable will be reinstated to the new item that matched. Note that the fourth attribute, whether the item is frozen or not, is irrelevant at this point. The third condition is a call to **bag-status**, which prevents us from putting too many items in one bag. The fourth, fifth, and sixth conditions are used to increment the number of items currently in the bag. Since the item has now been placed in the bag, the seventh condition retracts from the data base the fact that the item is unbagged. Finally, the last condition is used to perform tail recursion. The rule will call itself recursively until there are no more large bottles to be bagged.

The second clause is exactly like the first, except for the fact that it does not check for the container to be a bottle. In this case, the second condition simply checks for a large item regardless of the container type. Note that the

second clause will execute only after there are no more large bottles to be bagged. This is because of the order in which the clauses are written. As long as there are unbagged large bottles, the first clause will succeed. When there are no more large bottles, the first clause will fail. Prolog will then attempt to resatisfy the goal by satisfying the conditions in the second clause, and it will succeed as long as there are unbagged large items.

The third clause in **bag-large-items** is what is referred to as a catchall clause. In case both the first and the second clause of the rule fail, this clause ensures that the rule will succeed, since it matches any call to this rule. The first two clauses will fail when there are no items to be bagged, either because there were none initially, or because they have already been bagged by **BAGGER**. The catchall clause has a different purpose in each case. In the case where initially there were no items to be bagged, its only purpose is to ensure that the rule will not fail. In the case where the items have been bagged by the program, its purpose is first, to ensure success, and second, to stop the recursion, and initiate the backtracking process.

The rule **start-new-bag** is used to start a fresh bag after all the large items have been bagged. The first clause will succeed if there is an unbagged small item. However, we must prevent the rule from starting a fresh bag if there were no large items in the grocery list (remember that **init-bag** started a fresh bag). If the bag index is greater than one, this means that there were large items, and the first clause succeeds, but if the bag index is equal to one, then we must make sure that the number of items in the bag is not zero. The second clause takes care of this. If there was at least one large item bagged, and there is at least one small item to be bagged, then one of the first two



clauses should succeed. The third clause should execute only if there were no large items bagged, or if there are no small items to be bagged. The rule is shown below:

```
start_new_bag :- item(Name, _, small, _),
                 unbagged(Name),
                 bag_index(I),
                 I \= 1,
                 retract(bag_index(I)),
                 retract(num_items(N)),
                 J is I + 1,
                 asserta(bag_index(J)),
                 asserta(num_items(O)),
                 nl, print('Bag ', J), nl.
start_new_bag :- item(Name, _, small, _),
                 unbagged(Name),
                 num_items(N),
                 N \= 0,
                 retract(bag_index(I)),
                 retract(num_items(N)),
                 J is I + 1,
                 asserta(bag_index(J)),
                 asserta(num_items(O)),
                 nl, print('Bag ', J), nl.
start_new_bag.
```

The rule **bag-small-items** is similar to **bag-large-items**. The variable **Yes-no** will be instantiated to either yes or no, depending on whether the item is frozen or not. This variable will then be used by the rule **print-small** for printout purposes. If the item is frozen, then the rule prints out the name of the item and a message indicating that the item must be put in an insulated freezer bag. Otherwise, the rule will simply print out the name of the item. Both rules are listed below:

```

bag_small_items :- unbagged(Name),
                  item(Name, _, small, Yes_no),
                  bag_status,
                  retract(num_items(N)),
                  M = N + 1,
                  asserta(num_items(M)),
                  retract(unbagged(Name)),
                  print_small(Name, Yes_no),
                  bag_small_items.

bag_small_items.

print_small?(Name, no) :- write("      ", Name), nl.
print_small(Name, yes) :- write("      ", Name,
                               " in insulated freezer bag"), nl.

```

The final rule, **completed-bagging**, is shown below:

```

completed_bagging :- not(no_items),
                    retract(bag_index(I)),
                    retract(num_items(N)),
                    nl, print('Bagging completed!'), nl.

completed_bagging.

```

The first condition in the first clause checks that there were indeed items to be bagged at one point. If the grocery list was empty, the clause will fail, and the second one will execute. The second and third conditions in the first clause retract information which is no longer necessary from the data base, and the fourth condition prints the final message.

#### 9.4.4 Execution of the Prolog BAGGER

Figure 5 shows a sample list of groceries used as input to the Prolog program, and figure 6 shows the output generated by the program.

### 10 An Additional Example

In this section, we develop a simple program which helps a person to guess a sequence of numbers. The problem itself seems pretty useless, but it is intended to demonstrate at least some aspects of a problem for which OPS5 would be better suited than Prolog.

```
unbagged(granola).
unbagged(peanut_butter).
unbagged(ice_cream).
unbagged(soda).
unbagged(bread).
unbagged(granola).
unbagged(soda).
unbagged(potato_chips).
unbagged(soda).
unbagged(granola).
unbagged(soda).
unbagged(tv_dinner).
unbagged(soda).
unbagged(peanut_butter).
unbagged(ice_cream).
```

Figure 5: Sample list of input for the Prolog program

```
Bag 1
  soda
  soda
  soda
  soda
  soda
  granola

Bag 2
  granola
  granola

Bag 3
  peanut_butter
  ice_cream in insulated freezer bag
  bread
  potato_chips
  tv_dinner in insulated freezer bag
  peanut_butter

Bag 4
  ice_cream in insulated freezer bag

Bagging completed!
```

Figure 6: Output of the Prolog implementation of BAGGER

## 10.1 The Problem Description

The objective of the program is to aid the user in guessing a correct sequence of three one digit numbers. Figure 7 shows a tree which depicts all the possible correct sequences. A sequence is considered to be correct if there exists a path from the root to a leaf which follows that sequence.

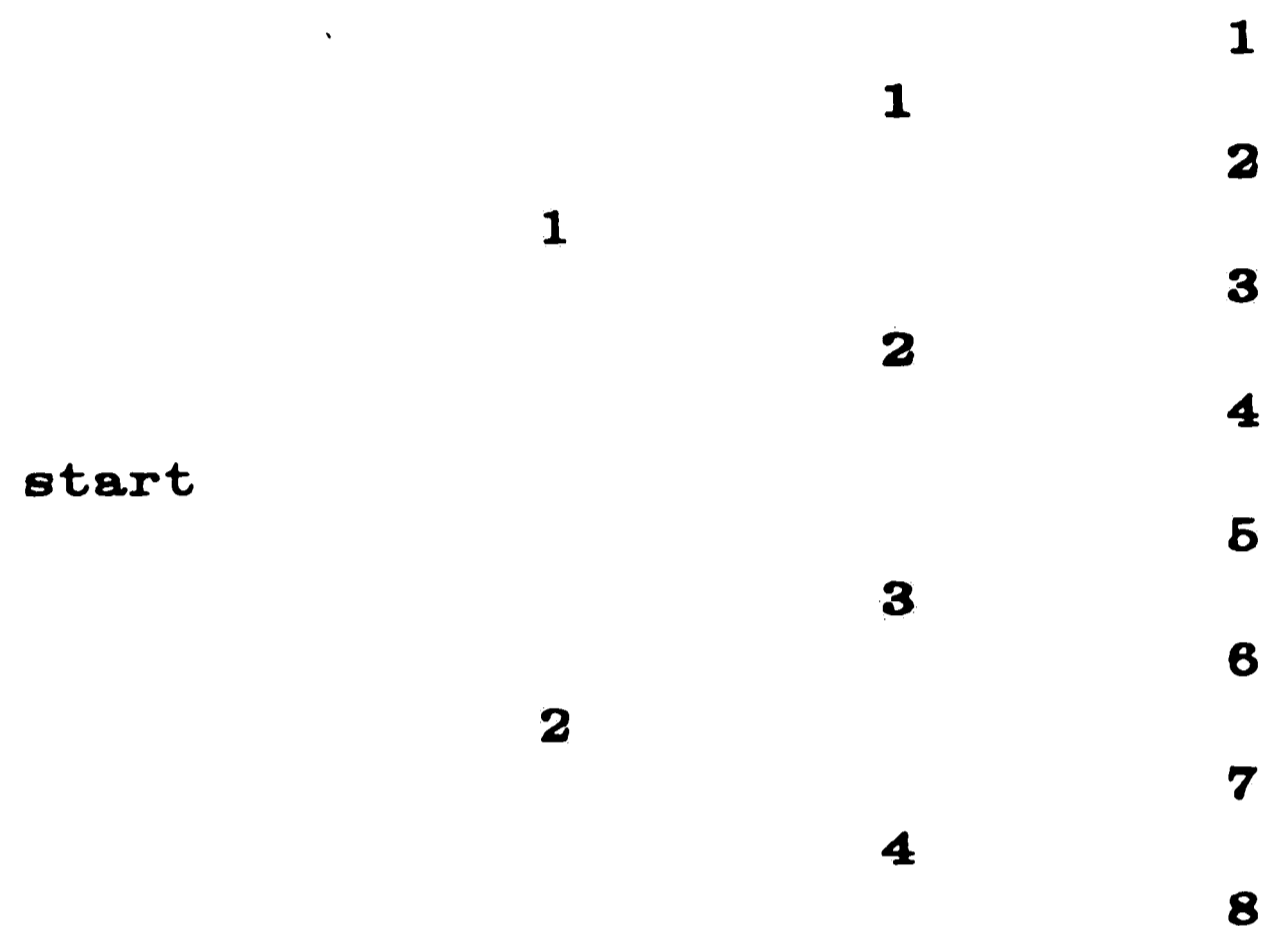


Figure 7: Binary tree depicting acceptable codes.

The program begins by printing the line

Please enter a 1 or a 2:

The user can then choose to type in the number 1, or the number 2. If he makes a mistake, and types in any other number, the program will inform him of his error, and prompt him for the number again. Once a correct entry has been made, the program will prompt the user for the next number in the sequence, always giving him a choice of correct numbers to select from. The program continues until the user has typed in a correct sequence of numbers.

This little program may seem trivial, and useless, but its main objective is analogous to that of an expert system: to guide the user through the search space, so as to make it easier for him to find a solution to his specific problem. The problem is also intended to simulate a real life problem which possesses

similar characteristics, namely a fan out situation, where we have one initial state, and numerous final and uncertain goals. We begin solving the problem in the initial situation, where we have a number of options to choose from. The option we choose will take us to a new situation, where we will then have a whole new set of different options. The process continues until we reach one of the goal situations. Note that, in any situation, we could have any number of decisions to make. The fact that, in this case, the tree is binary is a completely arbitrary decision.

Every node in the tree represents a state in a search space. At every state, we can choose different paths to follow which will take us to a state deeper in the space. Since we have a fan out situation, the problem solving method to be used is forward chaining. In this case, it would be feasible to use backward chaining because there are only eight goal states, but in other situations, the fan out could be much too large, or the goals too uncertain, to even consider using backward chaining.

Although the situation does not arise here, it is possible that two or more different paths could lead to the same state. In other words, two or more nodes in the tree actually represent the same node. The search which emanates from those nodes is exactly the same, and will be duplicated in the tree. A program, however, should be smart enough to detect this. All the nodes which represent the same situation, should be merged, so that the search can continue from that point, thus avoiding the need for repetitious code.

Since, as we said before, the above situation does not arise, we must create it artificially. We arbitrarily choose node 123 to be the same as node 2. In other words, if the user types in the sequence 1-2-3, it is just as if had

typed only the number 2 as the first number, and the search must continue from that point. The program must then prompt the user for the second and third numbers once again. Figure 8 shows the tree with this addition.

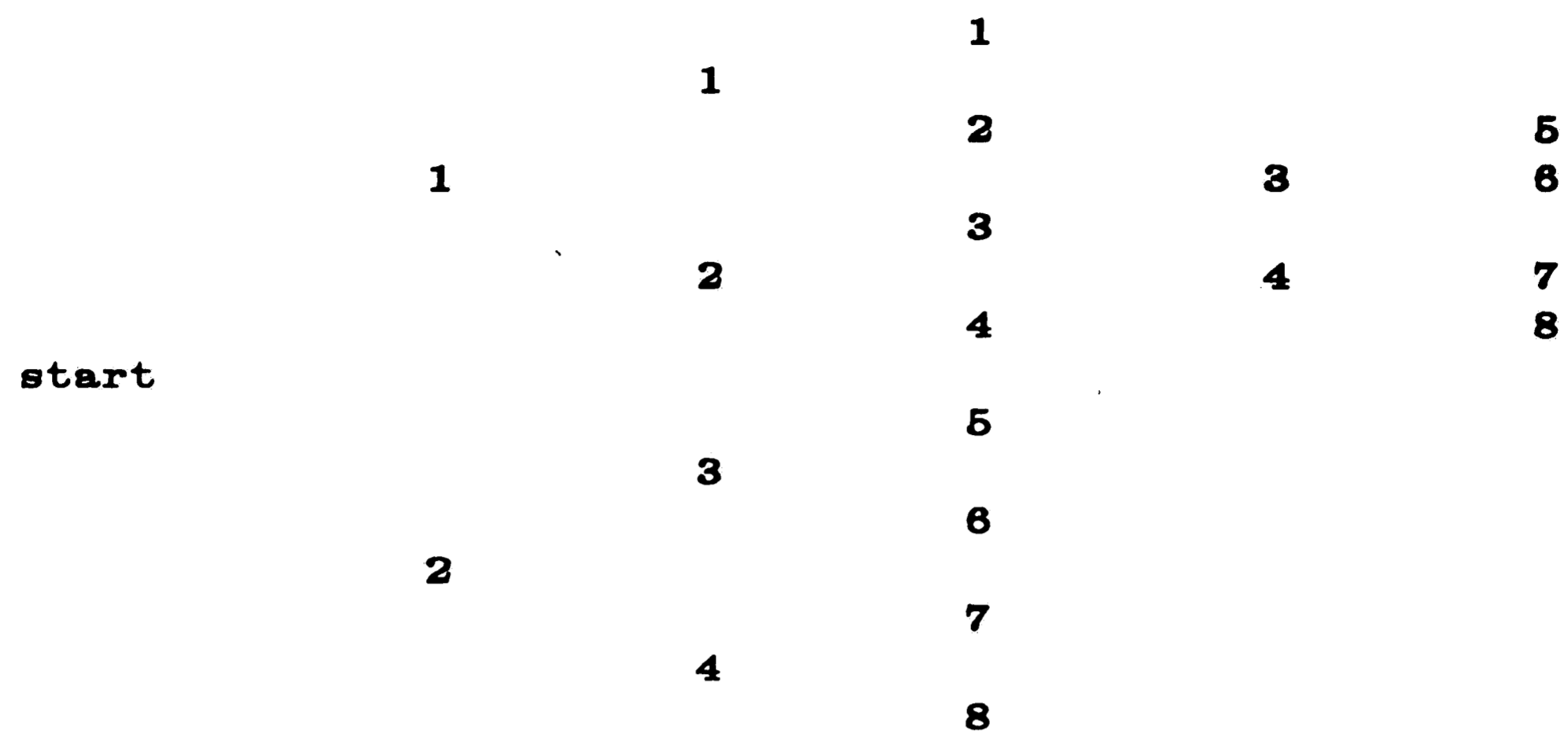


Figure 8: Tree With Additional Nodes

## 10.2 OPS5 Implementation of the Number Codes Example

The OPS5 program consists of four groups of rules. The first group contains only one rule whose purpose is merely to initiate execution. We begin with working memory containing the following element:

```
( step get_1st_number )
```

Rule **get-1st-number**, which is the rule which must fire first, checks for this element, and then prompts the user for the first number in the sequence. The third action clause demonstrates the way in which OPS5 accepts input interactively. The **accept** command causes a prompt to be displayed at the screen, and will wait for an atom to be entered. The **bind** command will bind the variable **<num1>** to the value of the atom just entered. The last action clause inserts into the database the fact that the first number entered was the value of the variable **<num1>**. The rule is shown below:

```

(p get_1st_number
  (step get_1st_number )
-->
  (remove 1 )
  (write (crlf) (crlf) |First number: please enter a 1 or a 2| )
  (bind <num1> (accept) )
  (make number 1 <num1> )
)

```

The second group contains all the rules whose name is only one alphanumeric character in length. Rule 1 will fire if the first number entered was a 1. By looking at the tree, we know that a 1 can be followed only by another 1, or a 2, so the rule prompts the user for either of these two numbers. The rule, however cannot check that the user will actually type in one of these numbers. At this point the program can only accept any number that the user wishes to type. Verification of correct input will be performed later on by another rule. The number that the user entered is stored in the data base as the second number in the sequence. The rule follows:

```

(p 1
  (number 1 1 )
-->
  (write (crlf) (crlf) |Second number: please enter a 1 or a 2| )
  (bind <num2> (accept) )
  (make number 2 <num2> )
)

```

Rule 2 will fire if the first number in the sequence was a 2. It prompts the user for either a 3 or a 4. Then it stores the number entered by the user into the data base.

```

(p 2
  (number 1 2 )
-->
  (write (crlf) (crlf) |Second number: please enter a 3 or a 4| )
  (bind <num2> (accept) )
  (make number 2 <num2> )
)

```

If the first number entered was neither a 1 nor a 2, then rule X fires. It

deletes the first number from the data base, since it is not a correct entry. It then prints a message informing the user that his entry was in error, and finally, it makes an entry to working memory which will enable the rule **get-1st-number** to fire again. After this rule fires, the elements in working memory are exactly the same as they were initially. The rule follows:

```
(p X
  (number 1 <num> )
-->
  (remove 1 )
  (write | Sorry, bad start. Try again. | )
  (make step get_1st_number )
)
```

The third group contains all the rules whose name is two alphanumeric characters in length. Their role is to verify that the second number entered is allowed to follow the first, and then to prompt the user for the appropriate third number. For example, Rule **23** will fire if the first number entered was a 2, and the second number entered was a 3. From the tree, we know that the only two numbers which can follow are 5 and 6, so the user is prompted to enter one of them. The rules in the group are shown below:

```
(p 11
  (number 1 1 )
  (number 2 1 )
-->
  (write (crlf) (crlf) |Third number: please enter a 1 or a 2| )
  (bind <num3> (accept) )
  (make number 3 <num3> )
)
```

```
(p 12
  (number 1 1 )
  (number 2 2 )
-->
  (write (crlf) (crlf) |Third number: please enter a 3 or a 4| )
  (bind <num3> (accept) )
  (make number 3 <num3> )
)
```



```

(p 23
  (number 1 2 )
  (number 2 3 )
-->
  (write (crlf) (crlf) |Third number: please enter a 5 or a 6| )
  (bind <num3> (accept) )
  (make number 3 <num3> )
)

(p 24
  (number 1 2 )
  (number 2 4 )
-->
  (write (crlf) (crlf) |Third number: please enter a 7 or an 8| )
  (bind <num3> (accept) )
  (make number 3 <num3> )
)

(p XX
  (number 1 <num1> )
  (number 2 <num2> )
-->
  (remove 2 )
  (write | Sorry, that number does not follow. Try again.| )
)

```

Rule **XX**, above is like a catchall rule, which will fire if none of the other rules in the group was able to fire. At this point, we know that the first number entered was correct (otherwise we wouldn't be this far down), so the rule removes the second number entered from working memory. After this rule fires, it will enable one of the rules in the second group to fire. The user will be prompted for the second number once again, and execution will proceed.

Production rule **123** is contained in the fourth group, but it is different from the rest because it deals with the special case where one node is the same as another. Since node **123** is the same as node **2**, and rule **2** fires when the search is at node **2**, we must create all the conditions necessary for rule **2** to fire. This is a simple thing to do, given the data driven nature of OPS5. By changing the appropriate working memory elements, it is possible for one production rule to enable another rule. This is exactly what rule **123** does. It creates all the necessary conditions for rule **2** to fire. It modifies the first

number in the sequence to be 2, and it deletes the other two entries. It is listed below:

```
(p 123
  (number 1 1 )
  (number 2 2 )
  (number 3 3 )
-->
  (modify 1 ^3 2 )
  (remove 2 )
  (remove 3 )
  (write (crlf) (crlf) |The code is one-two-three, | )
  (write |but you must return to node 2.| )
)
```

The rest of the rules in the fourth group should require no further explanation:

```
(p 111
  (number 1 1 )
  (number 2 1 )
  (number 3 1 )
-->
  (write (crlf) (crlf) |The code is one-one-one.| )
  (halt)
)
```

```
(p 112
  (number 1 1 )
  (number 2 1 )
  (number 3 2 )
-->
  (write (crlf) (crlf) |The code is one-one-two.| )
  (halt)
)
```

```
(p 124
  (number 1 1 )
  (number 2 2 )
  (number 3 4 )
-->
  (write (crlf) (crlf) |The code is one-two-four.| )
  (halt)
)
```

```
(p 235
  (number 1 2 )
  (number 2 3 )
  (number 3 5 )
-->
  (write (crlf) (crlf) |The code is two-three-five.| )
  (halt)
)
```

```

(p 236
  (number 1 2 )
  (number 2 3 )
  (number 3 6 )
-->
  (write (crlf) (crlf) |The code is two-three-six.| )
  (halt)
)

(p 247
  (number 1 2 )
  (number 2 4 )
  (number 3 7 )
-->
  (write (crlf) (crlf) |The code is two-four-seven.| )
  (halt)
)

(p 248
  (number 1 2 )
  (number 2 4 )
  (number 3 8 )
-->
  (write (crlf) (crlf) |The code is two-four-eight.| )
  (halt)
)

(p XXX
  (number 1 <num1> )
  (number 2 <num2> )
  (number 3 <num3> )
-->
  (remove 3 )
  (write | Sorry, no code available.  Try again.| )
)

```

### 10.3 Prolog Implementation of the Number Codes Example

In this section we discuss the Prolog implementation of the number codes problem. The Prolog program consists of three groups of rules. The first group contains only the main rule, which is used to initiate execution, and as the program driver. It is listed below:

```
start :- readfirst(X), readsecond(X, Y), readthird(X, Y, Z), !.
```

The second group contains three rules. Their purpose is mainly to read in the number that the user inputs, and then to check that the number follows in the sequence. The auxiliary rule `mess` is used to print out a message

prompting the user for the next number in the sequence, and then to read the number input by the user. It is shown below:

```
mess(S,T,U) :- print('Please enter a ', S, ' or a ', T, ' '),
               read(U).
```

The first rule in the second group reads in the first number and checks that it is either a 1 or a 2. It is shown below:

```
readfirst(X) :- repeat, mess(1,2,X), check(X).
```

If the user enters a number other than 1 or 2, `check(X)` will fail, and Prolog will backtrack to the `repeat` predicate. This is a built in predicate which is used to prevent Prolog from backtracking out of a clause until all the goals to the right of the `repeat` have been satisfied. In this case, it will not let the program backtrack out until the user types in a 1 or a 2. The definition of the `repeat` predicate follows. For a more detailed explanation, see

[Clocksin & Mellish 84].

```
repeat.
repeat :- repeat.
```

The rules in the third group check that the input entered by the user is valid. The rule `readfirst` calls the rule `check` with only one argument. The rule will check for either a 1 or a 2. If this was indeed the input, one of the first two clauses succeeds. Otherwise the third clause, which is a catchall, executes and causes the rule to fail.

```
check(1).
check(2).
check(X) :- print(X, ' is a bad input, please try again.'),
            nl, fail.
```

The remaining rules in the second group are `readsecond` and `readthird`. After `readfirst` succeeds (which means that the first number entered by the user was valid), the main rule will call `readsecond`. The purpose of

**readsecond** is to check that the second number is valid. Similarly, the purpose of **readthird** is to check that the third number is valid. They are listed below:

```
readsecond(1,Y) :- print('Okay. '), nl, repeat, mess(1,2,Y),
                  check(1,Y).
readsecond(2,Y) :- print('Okay. '), nl, repeat, mess(3,4,Y),
                  check(2,Y).

readthird(1,1,Z) :- print('Okay. '), nl, repeat, mess(1, 2, Y),
                   check(1, 1, Y).
readthird(1,2,Z) :- print('Okay. '), nl, repeat, mess(3, 4, Y),
                   check(1, 2, Y).
readthird(2,3,Z) :- print('Okay. '), nl, repeat, mess(5, 6, Y),
                   check(2, 3, Y).
readthird(2,4,Z) :- print('Okay. '), nl, repeat, mess(7, 8, Y),
                   check(2, 4, Y).
```

To check that the second number is valid, the **check** rule with two arguments is used. They are shown below:

```
check(1, 1).
check(1, 2).
check(2, 3).
check(2, 4).
check(_, Y) :- print(Y, ' is a bad input, please try again. '),
               nl, fail.
```

Finally, to check that the third number is valid the **check** rule with three arguments is used. With two exceptions, they all print out the sequence of numbers that the user just entered, if the sequence was a valid one. The first exception is the clause that checks for the input 1-2-3. If this was the input, then rules **readsecond** and **readthird** are called once again. The second exception is the last clause which will execute if the third number did not follow in the sequence, and cause the rule to fail.

```

check(1, 1, 1) :- print('The code is 1-1-1.').
check(1, 1, 2) :- print('The code is 1-1-2.').
check(1, 2, 3) :- print('The code is 1-2-3, '), nl,
                 print('but you must return to node 2'), nl,
                 readsecond(2, YY), readthird(2, YY, ZZ).
check(1, 2, 4) :- print('The code is 1-1-1.').
check(2, 3, 5) :- print('The code is 1-1-1.').
check(2, 3, 6) :- print('The code is 1-1-1.').
check(2, 4, 7) :- print('The code is 1-1-1.').
check(2, 4, 8) :- print('The code is 1-1-1.').
check(_, _, Z) :- print(Z, ' is a bad input, please try again'),
                 nl, fail.

```

## 11 Debugging Tools

An important factor which should be taken into consideration before choosing a software tool is the environment that the tool provides for writing, debugging, and executing programs. Debugging tools play an important part in simplifying the process of writing and debugging programs. If used effectively, they can save the programmer a considerable amount of time in this process. Both Prolog and OPS5 provide a standard set of debugging features. However, bear in mind that, in general, the commands available depend on the particular implementation, and may differ slightly from the ones described in this section.

### 11.1 Prolog Debugging Tools

Prolog has a set of built-in predicates which allow the programmer to watch the program as it executes. These predicates are discussed in this section.

```

check(1, 1, 1) :- print('The code is 1-1-1.').
check(1, 1, 2) :- print('The code is 1-1-2.').
check(1, 2, 3) :- print('The code is 1-2-3, '), nl,
                 print('but you must return to node 2'), nl,
                 readsecond(2, YY), readthird(2, YY, ZZ).
check(1, 2, 4) :- print('The code is 1-1-1.').
check(2, 3, 5) :- print('The code is 1-1-1.').
check(2, 3, 6) :- print('The code is 1-1-1.').
check(2, 4, 7) :- print('The code is 1-1-1.').
check(2, 4, 8) :- print('The code is 1-1-1.').
check(_, _, Z) :- print(Z, ' is a bad input, please try again'),
                 nl, fail.

```

## 11 Debugging Tools

An important factor which should be taken into consideration before choosing a software tool is the environment that the tool provides for writing, debugging, and executing programs. Debugging tools play an important part in simplifying the process of writing and debugging programs. If used effectively, they can save the programmer a considerable amount of time in this process. Both Prolog and OPS5 provide a standard set of debugging features. However, bear in mind that, in general, the commands available depend on the particular implementation, and may differ slightly from the ones described in this section.

### 11.1 Prolog Debugging Tools

Prolog has a set of built-in predicates which allow the programmer to watch the program as it executes. These predicates are discussed in this section.

### 11.1.1 The trace and notrace predicates

The **trace** predicate is an exhaustive tracing feature. When this feature is set, every time Prolog attempts to satisfy a goal, it prints out information which allows the programmer to trace the execution of the program. The following messages are displayed as program execution progresses:

- **CALL** ( <goal name> ). This means that Prolog is currently trying to satisfy a goal whose name is <goal name>.
- **EXIT** ( <goal name> ). This message means that Prolog has satisfied <goal name>.
- **REDO** ( <goal name> ). This message is displayed when Prolog is attempting to resatisfy <goal name> immediately after it failed, or after backtracking from another goal which also failed.
- **FAIL** ( <goal name> ). When Prolog is unable to satisfy <goal name>, this message is displayed.

To turn off exhaustive tracing, the **notrace** predicate is used. This, however will not turn off spy points set by the **spy** predicate which is discussed subsequently.

### 11.1.2 The spy and nospy predicates

The **spy** predicate is used to trace the execution the particular predicates which the programmer is interested in. It is invoked by typing **spy(<predicate name>), group**, where <predicate name> can be either of the following:

- An atom. A spy point is put on all predicates with this atom, regardless of the number of arguments.
- A structure if the form <name>/<num>, where <name> is the name of the predicate to be spied, and <num> is the number of arguments that the predicate has. Spy points are set only on clauses whose name is <name> and which contain <num> arguments.



- A list of the form:

[<name<sub>1</sub>>/<num<sub>1</sub>>, <name<sub>2</sub>>/<num<sub>2</sub>>, ..., <name<sub>i</sub>>/<num<sub>i</sub>>]

To remove spy points, the `nospy` predicate is used with an argument of the same form.

### 11.1.3 The debugging predicate

The `debugging` predicate allows the programmer to see the spy points which are currently set.

## 11.2 OPS5 Debugging Tools

In this section the standard debugging features of OPS5 are described.

### 11.2.1 The run command

The `run` command can be used as a debugging aid by giving it an argument. The argument is an integer  $n$ , where  $n$  is the number of rules which will be allowed to fire before execution halts. Obviously, execution will halt before the  $n$  rules are fired if any of the conditions for terminating program execution are met.

### 11.2.2 The watch command

The `watch` command, in general, is used with an argument which is an integer from zero to two. The possible codes are:

- (watch 0): this is the default code. The system does not report any rule firings or changes to working memory.
- (watch 1): when this code is used, the system prints out the name of each production rule as it is being fired. It also prints out the time tags of the working memory elements associated with the fired rules.
- (watch 2): in addition to the information in level 1, the system prints a message every time a working memory element is added, modified, or deleted.

If the watch command is entered without an argument, the current watch code is displayed.

### 11.2.3 The cs command

The cs command is used to examine the contents of the conflict set. A list of all the triggered rules is displayed, along with the dominant instantiation. This is the rule which will be chosen to fire, unless production memory, working memory, or the conflict resolution strategy are modified.

### 11.2.4 The pbreak command

The pbreak command is used with a rule name as an argument, to set a break point at that rule. Whenever that rule fires, execution halts, and control is returned to the top level. The command will accept any number of rule names as arguments, including zero. In case the command is entered with no arguments, the system displays the break points which are currently set.

### 11.2.5 The back command

The back command takes an integer number less than 32 as an argument, and undoes the effects of that number of rule firings.

## 12 Concluding Remarks

We saw that in the two languages the methods for representing facts are very similar. The major differences are:

- the representation of the rules
- forward vs. backward chaining rules
- sequential vs. data driven execution

The I/O and arithmetic capabilities of both languages are fairly limited, but it seems like this depends more on the particular implementation of the

language than on the language itself.

For small scale systems, where the problem domain is well defined or well structured, there are no significant differences in performance between OPS5 and Prolog. The choice of one over the other would be based more on personal preference. If the problem requires, or is better solved by a backward chaining search, then Prolog would probably be a better choice. Although OPS5 is capable of performing backward chaining by setting goals (in much the same way as in the BAGGER example), Prolog can do it more naturally. It would seem that OPS5 would excel in the case where forward chaining is required, but for most small systems, Prolog can handle forward chaining just as well. For a case such as the one described in the number codes example of section 10, where the search requires node jumps in the search tree, the Prolog program cannot avoid the need for repetitive code. In a situation where the number of node jumps is large, or if after a node jump the search is long, OPS5 would be a better choice.

[Brownston, et al. 85] state that:

The power of OPS5 is usually hidden when it is used for small system applications. This power is most evident when it is applied to large, ill-structured problems. OPS5, then, should be used when the problem-solving environment is complex (that is, when there are many independent states in the domain, and variations are large and important), and when responses to it must be diverse and based on attention to many factors. Also, because of its data-driven nature, OPS5 is well suited for applications where the data change during the execution of the program, and it is essential that the program respond appropriately to these changes. The new data immediately become part of working memory, which is what controls the firing of the rules in production memory. This allows OPS5 programs to shift attention quickly, and react to the data in an apparently intelligent

fashion.

On the other hand, when the problem-solving environment is complex, but well defined, and there exist specific algorithms to handle the problem-solving task, then a sequential language, such as Prolog, is preferable.

Sometimes, a particular problem may be ill-defined at first. In this case, the programmer may want to use OPS5 primarily to gain some insight on the problem. This will help him develop the appropriate algorithms to solve the problem. Once the algorithms have been developed, it may turn out that they can be processed much more efficiently by a sequential or procedural type language. The program can then be rewritten in the procedural language.

## References

- [Brownston, et al. 85] Brownston, Lee, Farrell, Robert, Kant, Elaine, and Martin, Nancy.  
*Programming Expert Systems in OPS5.*  
Addison-Wesley, Reading, MA, 1985.
- [Clocksin & Mellish 84] Clocksin, W.F., and Mellish, C.S.  
*Programming in Prolog.*  
Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984.
- [Waterman 86] Waterman, Donald A.  
*A Guide to Expert Systems.*  
Addison-Wesley, Reading, MA, 1986.
- [Winston 84] Winston, Patrick H.  
*Artificial Intelligence.*  
Addison-Wesley, Reading, MA, 1984.

## Vita

Fabio Jose Urbina, the son of Fabio and Flora Urbina was born on November 6, 1962 in San Jose, Costa Rica. For his secondary education he attended the Lycee Franco-Costaricien, graduating in 1979. After attending the University of Costa Rica for one year, he came to Lehigh University, where he received his Bachelor of Science Degree in Computer Engineering in June of 1984. He began his graduate studies in Computer Science in the Fall of the same year.