lıс

THE DESIGN OF THREE INTERPRETERS: PROPOSITION, PROPOSITION PROOF, & PREDICATE

by

Sarah R. Lauxman

B.S., Kansas State University, 1985

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by

[illegible]
Major Professor

ACKNOWLEDGEMENTS

I would like to thank Dr. William Hankley for all his help, advice, and support in making the project and report a success.

Thanks to Dr. Elizabeth Unger and Dr. Austin Melton for their suggestions and moral support.

Special thanks to my husband, Douglas H. Lauxman, for his continued support, patience, and understanding.

Thanks to Karen Brewer for always reminding me that I would finish in time for graduation.

Thanks to Weilin Chen, Joe Vide, and other graduate colleagues for their support.

Thanks also to my family members for their moral contributions to this report.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

This report presents the design and implementation of a family of three related stack-machine interpreters, similar to the p-machine interpreted Pascal systems. The input for each interpreter is a series of pcodes. The pcodes represent single boolean expressions or series of boolean expressions. The interpreters execute the pcodes using test cases and evaluate the resulting boolean.

The scope of the pcodes includes operands, operators, and various labels. The operands are identifiers, integer constants, integer arrays, indices, and ranges. The operators are arithmetic, indexing, relational, logical, and quantifiers.

Each interpreter is part of a larger system, which is described as background information but which was not a part of the work of this project. The input for each interpreter, the pcodes, is produced by a parser/code-generator. The parser takes "programs" which are created by a user's interpretation of stories in the domain of logic problems, much like those found in logic text books. The story problems belong to one of three different problem domains (proposition, proposition proof, and predicate). These problem domains correspond to three languages named proposition, proposition proof, and predicate. Chapter 2 describes the problem domains.

```
 ________________    ---> (_) --->  ____________       ____________
|"all men        |        \|/      |all(x) :    |---> |produce     |--->pcodes
|are mortal"     |         |       |mortal(x)   |     | pcodes     |
|________________|        / \      |____________|     |____________|
Story Problem            User         Program           Parser

                         pcodes-->  ____________________
                                   |                    |
 _____________                     |  execute pcodes    |--->Interpreter
|             |                    |  using test cases  |      results
| x = John    |------------------> |       and          |
| x = Bob     |                    | evaluate boolean   |
| x = Don     |                    |  result            |
|_____________|                    |____________________|
Test Case                              Interpreter
  File
```

Figure 1.1 : Walk-through of a System

Figure 1.1 illustrates the sequence of steps that occur in the system. First the user chooses a story problem from one of the problem domains. Then the user creates a program to symbolize the story problem. The symbolized program is input to the parser/code generator which produces the pcodes for the respective interpreter. The interpreter then executes the pcode using test case values and evaluates the boolean result for each test case.

The user creates the programs in an external language defined by one of the three "source" grammars. The parser checks the program to make sure that it follows the syntax described in the source grammar. Each source grammar is described in conjunction with its corresponding problem domain discussion in Chapter 2. The code generator produces the pcodes using the definitions from one of three "target" grammars, again, one for each of the three interpreters. Each source and target grammar are related by a "translation" grammar. The three target grammars and the three translation grammars are discussed as Chapter 3.

Chapters 4, 5, and 6 are detailed discussions of the three languages, one chapter per language. Chapter 4 is the proposition language, chapter 5 is proposition proof, and chapter 6 is predicate. These three chapters have a parallel structure, each describing the external language, grammar, parser, and interpreter for the respective problem domain. The structure of these three chapters is the following one.

The internal language structure -- the pcodes

The pcodes for the language are described.

The test case file

The contents of the test case file are described. An explanation is given as to how to derive the values for the test cases.

The test case file structure

The structure of the test case files are illustrated.

The interpreter design

The interpreter design is described, including how the interpreter executes the pcodes using the test case values and how the resulting boolean value is evaluated.

The last chapter, the conclusion, discusses and illustrates some of the problems that were encountered in the design of the project.

Chapter 2

PROBLEM DOMAINS & SOURCE GRAMMAR DESCRIPTIONS

The larger project from which the work of this project was derived originated from the compiler class taught at Kansas State University in the spring semester, 1986. Class programming projects included building parsers and code generators for three similar grammars. Again, the parsers and code generators are not part of this report. Each grammar was for a different problem domain, which had been introduced in the programming science course at KSU.

The whole project is designed to aide students in writing programs for story problems in the programming science course. Since the story problems used in the project belong to one of three different problem domains covered in the class, the students have a means of exercising three of the skills taught in the course.

In the next three sections of this chapter each of the three problem domains are described. In each problem domain section, the corresponding source grammar is discussed. The fourth section describes all three translation grammars and all three target grammars together, since they have a similar conceptual structure.

Problem Domain : Proposition

For the proposition problem domain, the user is given a simple story problem such as "all dogs are animals". The user must composed a sequence of propositions using names defined in the story: "all(x) : animal(x)". A proposition is a boolean (logical) expression, comprised of logical constants, objects with predefined predicates, relational expressions on integers and on one-dimensional integer arrays. The relational expressions map to boolean values. Propositions use the operators not, and, or, implies, and equivalence. Propositions also use parentheses to determine the order of evaluation. Figure 2.1 lists six examples of propositions.

```
Logical Constant  :  true
Boolean Expression  :  k implies j
Predefined Construct  :  raining(cats) and raining(dogs)
Proposition with Parentheses  :  (a = b)
Relational Expression on Integers  :  x = y + 1
Relational Expression on Integer Arrays  :  B[i] < B[i+1]
```

Figure 2.1 : Proposition examples

The source grammar for the proposition problem domain is listed in Appendix A -- Source Grammars. The source grammar is context-free; however, the terminals in the source grammar that are "names" (the choices for the rules "Var", "Avar", "Ivar", "Pvar", and "Cvar") are restricted to being only the names used in the story problem. "Var" is any variable name used in the story problem, "Avar" is any array name, "Ivar" is any index, "Pvar" is any predefined construct, and "Cvar" is any constant. For example, if a story problem read: x is twice as large as y, then the only names that can be used in a program are "x" and "y". This constraint is enforced because the interpreter uses a name table, similar to a compiler's symbol table, to substitute different values for the variables of a story problem during the execution and evaluation of a program. Because of the name constraint, the source language is context-sensitive.

The proposition source grammar is the building block for the other two source grammars. The "props" and "prop" rules are unique to the proposition source grammar; however, the rest of the rules are included in each of the other two source grammars. The next few paragraphs explain some of the rules of the proposition source grammar.

"Props" represents a collection of "Prop"s, which are propositions. One proposition may be broken into two or more smaller propositions, each on a separate line, to make the program easier to read. If the proposition is broken into several lines, then, excluding the last one, all lines are identified by a label. The last line uses the labels instead of the actual expression. The entire "program", therefore, is easier to understand. Figure 2.2 illustrates this idea; 2.2a is the proposition as one line and 2.2b breaks it into four lines.

```
a.  x > y and y > z or x > z

b.  :one:   x > y
    :two:   y > z
    :three: x > z
            one and two or three
```

Figure 2.2 : Example proposition using labels

Problem Domain : Proposition proof

For the proposition proof problem domain the user is given a proof problem as in Figure 2.3a and is required to enter a sequence of steps which proves the conclusion similar to Figure 2.3b.

```
a.  Given : premise 1 : a implies b
            premise 2 : b implies c
            premise 3 : a = true
    Prove : b = true

b.  a implies c  // premise 1, premise 2
    b = true     // premise 3, fact 1
```

Figure 2.3 : Example proposition proof

The proof problems use axioms, premises, facts and a conclusion. Axioms are propositions that are taken to be true at face value -- without proof. Premises are propositions that are the "given" parts of a proof; they are taken to be true only for the proof in which they are used. Facts are steps of the proof consisting of a proposition and a list of reasons why. The reasons why consist of axioms, premises and previous facts. The conclusion is the proposition which the user is required to prove.

An example proposition proof story problem is pictured as Figure 2.4. Axioms, premises, and a conclusion are outlined in the story problem; the steps of the proof have been omitted.

```
axiom 1 :
    not(stmts_have_side_effects) implies preds_are_stmts
axiom 2 :
    not(preds_are_stmts) implies not(prgmers_write_pgms)
axiom 3 :
    not((prgmers_write_pgms) and not(preds_are_stmts))
       implies stmts_have_side_effects
axiom 4 :
    not(not(stmts_have_side_effects) or
    not(prgmers_write_pgms)) implies
    not(programmers_are_rich)
premise 1 :
    programmers_write_pgms
conclusion :
    not(programmers_are_rich)
```

Figure 2.4 : Proposition proof example

The proposition proof source grammar adds the use of axioms, premises, and facts to the proposition source grammar. With these additions, the rules "Prog", "Step", "Reasons", and "Reason" replace the "Props" and "Prop" rules in the proposition source grammar to create the proposition proof source grammar. The next few paragraphs explain these rules.

A proof consists of a sequence of steps which correspond to the facts used to prove the conclusion. The "Step" rule consists of a label (:"F"Int:) identifying the fact number of the proof, a logical expression ("Lexp"), zero or more "logical operator logical expression" groups, and the reasons ("Reasons") why the fact is true. "Prog" is one or more "Step"s.

"Reasons" is one or more "Reason". "Reason" is either an axiom number ("A"Int), a premise number ("P"Int), or a fact number ("F"Int). The axiom, premise, or fact number corresponds to the axiom, premise, or fact number used in the proof problem to represent the pcodes for the axiom, premise or fact respectively.

Problem Domain : Predicate

For the predicate problem domain the user is given a story problem, similar to the ones used in the proposition problem domain, and is required to enter a program that corresponds to the story problem. Statements in the language are propositions which may be composed using predicates. Figure 2.5 illustrates a simple story problem and program.

```
Story Problem : all values b[j..k] are greater than 25
Program : ( ForAll i : ( j..k ) : b[i] > 25 )
```

Figure 2.5 : Example predicate story problem and program

The statements are propositions with the addition of quantifiers and ranges. The quantifiers are "ThereExists", "ForAll", and "NumberOf". The general form of a predicate is

(quantifier : range : proposition). Three predicate examples are illustrated in Figure 2.6.

```
a. (ThereExists i : (j..k) : b[i] = 0)
b. (ForAll i : (j..k) : b[i] = 0)
c. (NumberOf i : (j..k) : (b[i] = 0) <= 5
```

Figure 2.6 : Predicate examples

For the quantifier "ThereExists", the meaning of the predicate would be there exists a value in the given range that makes the proposition true. In Figure 2.6a., the predicate reads there exist a value, i, in the range j to k such that array b element i equals zero.

For "ForAll", the meaning would be for all values in the given range the proposition is true. In Figure 2.6b., the predicate reads for all values, i, from j to k, array b element i equals zero.

Finally, for "NumberOf" a number expression is given as the number of values in the range that make the proposition true. Figure 2.6c. reads the number of values, i, from j to k, that satisfy array b element i equals zero is less than or equal to five.

The predicate language is a subset of first order logic with a few restrictions. The quantifiers are over integer variables only, such as (ForAll (i) : b[i] > 0). Another restriction is that the proposition is limited to relational or logical expressions using objects with predefined predicates, one-dimensional integer arrays, and integer constants.

The predicate source grammar needs rules to define the structure of a predicate and the use of the qualifiers. Exchange the "Prop" rule of the proposition source grammar with the rules "Prog" (different from the "Prog" rule of the proposition proof source grammar), "Pred", "Qualex", "Qual", and "Numex" to create the predicate source grammar. Note that the predicate source grammar uses the "prop" rule of the proposition source grammar. The next few paragraphs explain these new rules.

A "Prog" is zero or more labeled "Prop"s ending with a "Pred". "Pred" is represented by either the "Qualex" rule or the "Numex" rule, a relational operator ("Relop"), and an index expression ("Iexp").

Chapter 3

TARGET AND TRANSLATION GRAMMARS

The pcodes, output from a code generator, are the input for the respective interpreter. The pcodes are defined by one of three target grammars. Each language has a target grammar that defines its pcodes. The target grammars are context-free; but by enforcing the use of the names given in the story problems, by using the name table, the target languages are context-sensitive. The three target grammars are listed as Appendix C -- Target Grammars.

In the target grammars the terminating items are pcodes which are stack machine instructions. Each has three parts: operator, operand, operand; not all of the instructions show values for all three of these parts. A zero is the understood missing operands. In Appendix D - Opcode Table, each operator (opcode) is listed with its meaning and the steps performed by each interpreter.

Some of the stack machine instructions can be optimized by the code generator. For example, instead of pushing two operands onto the stack and then popping them off to perform a relational operation, the relational instruction would be comprised of the operator and both of its operands. The pcodes would be one instruction verses three.

All operands that end with "inx" are actually indices into a name table which is used in a similar fashion as a symbol table is used in a compiler.

The translation grammar for each language represents the relationship between each language's source and target grammars. Each translation grammar, therefore, illustrates how each target grammar is derived from each source grammar. Each of the three translation grammars are attribute grammars and are listed in Appendix B -- Translation Grammars.

The objects inside the curly brackets are either intermediate steps towards the pcodes or semantic actions. An example of an intermediate step is shown in the "Prog" rule of the predicate translator grammar. The { "olang 3" } is added to the front of the "Prog" rule of the predicate source grammar and { "oend" } is added to the end which creates the "Prog" rule of the predicate translation grammar. These two intermediate steps are replaced by the terminals "olang", "3", and "oend" in the predicate target grammar.

Some of the items in the curly brackets are semantic actions. Semantic actions are distinguished from the intermediate steps by the use of the symbols "<--" which means replace the left side with the value on the right side. The replace action is the semantic action. For example, in the "Relop" rule, the first choice is to replace the "rel" with the operator "oeq". The replacement takes place in the rule "Rexp" even though it is described in the "Relop" rule.

Some of the attributes in each of the three translation grammars use the suffix "inx". "Inx" is also a constraint of these grammars because it corresponds to an index into the name table.

To understand how a program is translated into pcodes, an example problem from the proposition problem domain is illustrated in Figure 3.1. The program is "x = z". A derivation tree for each of the grammars (source, translation, and target) is drawn for the program and the resulting pcodes are given in Figure 3.1.

```
Program  :  " x = y "

Source grammar:        Translation grammar:

Props                  Props
|                      |
Prop                   {"olang 1"}
|                      Prop    {"oend"}
Lexp                   |
|                      Lexp
Rexp                   |
|                      Rexp
Exp Relop Exp          |
|   |     |            Exp Relop             Exp {"oeq"}
Var =     Var          |                     |
|         |            Var {"opush xinx"}    Var {"opush zinx"}
 x         z


Target grammar:                              Pcodes:

Props                                        olang 1    0
|                                            opush xinx 0
"olang" "1"                                  opush zinx 0
Prop "oend"                                  oeq   0    0
|                                            oend  0    0
Lexp
|
Rexp
|
Exp                Exp                Relop
|                  |                  |
"opush" "xinx"     "opush" "zinx"     "oeq"
```

Figure 3.1 :
Grammar Derivation trees and pcodes for a proposition example

Chapter 4

PROPOSITION LANGUAGE

The internal language structure -- the pcodes

The pcodes operators for the proposition interpreter are listed in the first table of Appendix D: "Proposition Opcode Table". In the table, the operator column lists the names of all the operators. The meaning column explains what each operator represents and the interpreter code column describes the action executed for each operator by the interpreters.

The first pcode for the proposition language is "olang 1 0" and the last one is "oend 0 0". The operands are either indices into the name table, "0", or an immediate operand value; a "1" is used in the first pcode represents the fact that the pcodes represent a program for the proposition language. The name table keeps track of variable names, values, and types. The types correspond to the source grammar rules "Var", "Avar", "Int", etc. For example, if the variable is an array name, then from the source grammar it is an Avar and the name table type is "array".

A "0" means either the value is on the top of the run stack or no value is used. Some operators only need one operand, such as olang and opush, while some operators do not require operands, such as oend and owhy.

Table 4.1 displays the pcodes for the example: "x > y and y > z or x > z". The operands "10", "11", and "12" in the table are indices into the name table. These indices correspond to x, y, and z respectively. The operands for "oand" and "oor" are on the run stack; therefore, "0"s are used in the pcodes for the operands for "oand" and "oor". The zero operands for "oend" and the zero operand for "olang", are place holders for the missing operands as defined previously.

Table 4.1 : Proposition sample pcodes

Operator	Operand	Operand
olang	1	0
ogrt	10	11
ogrt	11	12
oand	0	0
ogrt	10	12
oor	0	0
oend	0	0

The test case file

In testing a program for a story problem in the proposition problem domain, the number of test cases needed is determined by the number of primitive logical expressions used in the solution program. The primitive logical expressions are either relational expressions or predefined predicates. The number of test cases is equal to two raised to the power of "the number of primitive logical expressions used in the solution program". Therefore, if the solution program used one primitive logical expression, then two test cases are formed; two primitive logical expressions -- four test cases, three -- eight test cases, etc.

A table is created similar to the one in Table 4.2. Each relational expression is represented in the table and the true/false combinations are listed under the relational expression headings. For the example program "x > y and y > z or x > z", three primitive logical expressions were used, namely three greater than operators; therefore, eight test cases are needed.

Table 4.2 : Table used to derive test case file

(x>y)	(y>z)	(x>z)	x y z	result
true	true	true	3 2 1	true
true	true	false	- - -	----
true	false	true	3 1 2	true
true	false	false	2 1 3	false
false	true	true	2 3 1	true
false	true	false	1 3 2	false
false	false	true	- - -	----
false	false	false	1 2 3	false

Each variable in the story problem is represented as a heading in the table just as x, y, and z are placed in Table 4.2 for the example program. Values are selected for the variables to satisfy each row of the table. For example, if 3, 2, and 1 are substituted for x, y, and z respectively, then the first row which is "true true true" is satisfied. The rest of the table is completed in the same manner and the resulting sets of values for the variables are the test cases.

The last column of a table holds the value of each test case. A test case value is derived by substituting true or false for each relational expression and evaluating the program.

The test case file for the proposition language has a test case for each possible combination of true/false values for the relational expressions for one program solution for a given story problem. A user may create a different program for the story program. For example, using the story problem "y is less than or equal to z" the following three programs are correct: "x >= y", "y < x or x = y", and "x >= y or y < x".

A user cannot create a correct program using less than number of relational expressions as used in the solution program. Therefore, executing the pcodes, which represent the program, using the values from each test case is a sure method of verifying that the program from the user is(is not) a correct interpretation of the story problem.

Some programs in the proposition language will not have the exact number of test cases as described in the first paragraph of the previous section. The reason behind a test case file's shortage of test cases is for some programs, one or more of the rows of the table cannot be satisfied. For instance, in the example problem only six cases are possible because a case cannot be written where (x>y) and (y>z) are true while (x>z) is false nor can one be written where (x>y) and (y>z) are false while (x>z) is true.

The test case file structure

The test case file for the proposition interpreter has for each test case, a value for each variable in the story problem. After each variable is represented for a test case, the value of the test case is listed. The value of the test case is either true or false; true is represented by a "1", false by a "0".

Each item in the test case file starts in column one of the file and resides on a separate line from the rest of the items. For the variables of the test cases the name of the variable is first followed by its test value. The value of each test case (1 for true or 0 for false) is inserted at the end of each test case. The symbol "&" is used as a signal to the interpreter that the test case value is next in the file. The test cases are placed in the file one after another with absolutely no blank lines. For the example "x > y and y > z or x > z" the test case file is illustrated in Figure 4.1.

```
x CR  3 CR    y CR  2 CR    z CR  1 CR    & CR  1 CR
x CR  3 CR    y CR  1 CR    z CR  2 CR    & CR  1 CR
x CR  2 CR    y CR  1 CR    z CR  3 CR    & CR  0 CR
x CR  2 CR    y CR  3 CR    z CR  1 CR    & CR  1 CR
x CR  1 CR    y CR  3 CR    z CR  2 CR    & CR  0 CR
x CR  1 CR    y CR  2 CR    z CR  3 CR    & CR  0 CR
```

Figure 4.1 : Proposition test case file example

For the first test case in Figure 4.1, the values 3, 2, and 1 are substituted for x, y, and z respectively, yielding "3 > 2 and 2 > 1 or 3 > 1". After examining the new expression, it is easy to see that the test case result is true (1). The rest of the test cases can be explained in the same fashion.

The interpreter design

The proposition interpreter executes the pcodes using the values from each test case and evaluates the true/false result for each test case. All test cases must produce correct results in order for the user's program to be a correct interpretation of the story problem. For instance, from the test case file section of this chapter, the following story problem has six test cases: "either x is greater than y and y greater than z or x is greater than z". All six test cases must yield correct results before the user's program is accepted.

The interpreter places the first set of values from Table 4.2 (values for x, y, and z) into the name table as the values for x, y, and z respectively. Then the pcode is executed for a true/false result which is compared to the true/false value (the value after the "&" symbol) in the test case. If these two boolean values do not match then the evaluation of the program stops and the following message is returned to the user: "your program is incorrect". If, however, the values do match then the next test case is executed and evaluated. If each test case is successful, then the following message is returned: "your program is correct".

```
Begin
   error <-- false
   Repeat
      store test case values
      execute pcodes using test case values
      If result <> test case result Then
         error <-- true
      Endif
   Until eof(test case file) or error
   If error Then
      output "your program is incorrect"
   Else
      output "your program is correct"
   Endif
End
```

Figure 4.2 : Proposition interpreter algorithm

Figure 4.2 illustrates the algorithm used for the proposition interpreter. The code for the interpreter reads a test case, substitutes the values into the name table, executes the pcodes using the previously set values from the name table, compares the executed result against the correct one from the test case file, and then either continues with the next test case or quits because of the evaluation of the result is negative or the last case has been executed.

Chapter 5

PROPOSITION PROOF LANGUAGE

The internal language structure -- the pcodes

The "pcodes" for the proposition proof interpreter are described in Appendix D in the second table: "Proposition Proof Opcode Table". The first pcode is "olang 2 0" and the last one is "oend 0 0".

Table 5.1 illustrates the pcodes for the example proof in Figure 5.1. The operands "a1inx", "a2inx", "p1inx", "f1inx", "cinx", "tominx", "takeinx", "goinx", and "eatinx", listed in the table, are indices into the name table. These indices map to axiom 1, axiom 2, premise 1, fact 1, conclusion (fact 2), tom, takebus(tom), gohome(tom), and eatdinner(tom), respectively, in the problem outlined in Figure 5.1. At run time, the values for the operands for the operator "oimp", for this example, are at the top of the stack; therefore, in the pcodes the operands are "0".

```
Problem:
  axiom 1     : takebus(tom) implies gohome(tom)
  axiom 2     : gohome(tom) implies eatdinner(tom)
  premise 1   : takebus(tom)
  conclusion  : gohome(tom)


Solution:
   fact 1 :
    takebus(tom) implies eatdinner(tom) // axiom 1, axiom 2
   fact 2 :
    conclusion                          // premise 1, fact 1
```

Figure 5.1 : Proposition proof story problem & program

The pcodes between the asterisk lines in Table 5.1 are the axioms, premises and conclusion. The pcodes for the axioms and premises are saved and later executed as the pcodes for the reasons used for the facts in the user's proof program. For example, in the proof of Figure 5.1, the first reason of the first fact is "axiom 1". The pcodes which are executed for "axiom 1" are those between the pcodes "oaxiom a1inx 0" and "oaxiom a2inx 0" in Table 5.1. The asterisk lines are not part of the pcodes.

Table 5.1 : Proposition proof example pcodes

Operator	Operand	Operand
olang	2	0
**********************		
oaxiom	a1inx	0
oarg	tominx	0
ofun	takeinx	0
oarg	tominx	0
ofun	goinx	0
oimp	0	0
oend	0	0
oaxiom	a2inx	0
oarg	tominx	0
ofun	goinx	0
oarg	tominx	0
ofun	eatinx	0
oimp	0	0
oend	0	0
oprem	p1inx	0
oarg	tominx	0
ofun	takeinx	0
oend	0	0
ofact	cinx	0
oarg	tominx	0
ofun	goinx	0
oend	0	0
**********************		
ofact	f1inx	0
oarg	tominx	0
ofun	takeinx	0
oarg	tominx	0
ofun	eatinx	0
oimp	0	0
owhy	0	0
oaxiom	a1inx	0
oaxiom	a2inx	0
oend	0	0
ofact	cinx	0
owhy	0	0
oprem	p1inx	0
ofact	f1inx	0
oend	0	0
oend	0	0

The test case file

In testing a program for a story problem in the proposition proof problem domain, the number of test cases needed is calculated the same way as for the proposition problem domain. For the sample problem, three primitive logical expressions are used: gohome(tom), takebus(tom), and eatdinner(tom), so eight test cases are needed.

Table 5.2 : Proposition proof test cases for example problem

takebus(tom)	gohome(tom)	eatdinner(tom)
true	true	true
true	true	false
true	false	true
true	false	false
false	true	true
false	true	false
false	false	true
false	false	false

For the proposition proof test case file, a table is created similar to the one for the sample problem in Table 5.2. Each primitive logical expression of the problem is listed as a heading of the table and a truth table is created below the headings. Each line of the table represents one test case.

The test case file structure

The test case file for the proposition proof interpreter has for each test case a true/false value for each primitive logical expression used in the story problem. The true values are represented by a "1" and false by a "0" just as they were in the proposition language.

Again, each item is on a separate line starting in column one and the file has no blank lines. The variable name is listed first and the value next; therefore, each test case consists of the name and corresponding truth value of every variable used in the story problem. For the example described in Figure 5.1, the test case file would appear as Figure 5.2.

```
takebus CR  1 CR     gohome CR  1 CR     eatdinner CR  1 CR
takebus CR  1 CR     gohome CR  1 CR     eatdinner CR  0 CR
takebus CR  1 CR     gohome CR  0 CR     eatdinner CR  1 CR
takebus CR  1 CR     gohome CR  0 CR     eatdinner CR  0 CR
takebus CR  0 CR     gohome CR  1 CR     eatdinner CR  1 CR
takebus CR  0 CR     gohome CR  1 CR     eatdinner CR  0 CR
takebus CR  0 CR     gohome CR  0 CR     eatdinner CR  1 CR
takebus CR  0 CR     gohome CR  0 CR     eatdinner CR  0 CR
```

Figure 5.2 : Proposition proof test case file example

The interpreter design

The pcodes for the proposition proof language include pcodes for the axioms, premises, and conclusion, and as well as the pcodes for each fact. The pcodes for each axiom, premise, and the conclusion are executed several times by the interpreter, therefore, the respective pcodes are stored in an array (this array is called "fact-array" in the report). Facts that the interpreter accepts are also placed in fact-array.

The pcodes in fact-array for the axioms start with the pcode: "oaxiom ainx 0". The premises start with "oprem pinx 0", facts -- "ofact finx 0" and the conclusion -- "ofact cinx 0". The pcodes for the axioms, premises, and facts end with the pcode: "oend 0 0". The entries are placed end-to-end in fact-array; to locate a particular axiom, premise or fact a linear search is performed.

Each reason for a fact is represented in fact-array; the interpreter, therefore, searches the pcodes in fact-array for the one with the same operator (oaxiom, oprem, ofact) and first operand (ainx, pinx, finx, cinx) as the reason given for a particular fact. The pcodes from that position in the array until the "oend 0 0" pcode are then executed as the reason.

Table 5.3 : Table 5.2 with execution results (True results lists the axioms, premises, and facts that are true for each test case.)

Case	takebus(tom)	gohome(tom)	eatdinner(tom)	True Result
1	true	true	true	a1,a2,p1,f1,f2
2	true	true	false	a1, p1 ,f2
3	true	false	true	a2,p1,f1
4	true	false	false	a2,p1
5	false	true	true	a1,a2, f1,f2
6	false	true	false	a1, f1,f2
7	false	false	true	a1,a2, f1
8	false	false	false	a1,a2, f1

The proposition proof interpreter begins by loading the pcodes for the axioms, premises, and conclusion into fact-array. Then each fact is executed by the following method. For each test case, the reasons are executed and the resulting boolean value is evaluated. If all reasons produce true results for a test case, the fact is executed using the same test case values. If the fact also evaluates to true, then the testing continues with the next test case, else execution stops and the fact is rejected. If any reason evaluates to false, the interpreter starts again with the next test case and the first reason.

For example, for the first test case in the sample problem, the value "true" would be substituted as the value for takebus(tom), gohome(tom), and eatdinner(tom) in the name table. The interpreter takes the first reason given for the first fact and executes the pcodes for it using the values of the first test case. Using the "True Result" column of Table 5.3, which lists the axioms, premises, and facts that evaluate to true, the result of executing axiom 1 (the first reason for the first fact) is true. The next reason, axiom 2, is executed using the first test case values and the result is true too. Axioms 1 and 2 are the only reasons for fact 1 and each were true for the first test case so the fact pcodes are executed. The result is again true; therefore, a counting variable, TrueCount, initialized to zero, is incremented.

Next, the interpreter places the values of the second test case into the name table and the above execution pattern is performed again. From Table 5.3, only axiom 1 is true so the next test case is loaded into the name table. This case fails because of axiom 1 as does the fourth test case. The fifth one, however, has a true result for axiom 1, axiom 2, and the fact so TrueCount is again incremented. The next test case fails since axiom 2 is false. The last two result in true for both the axioms and the fact so TrueCount is left with the value of four. The first fact is accepted as the value of TrueCount is greater than zero.

The interpreter then executes next fact, the conclusion, in a similar fashion starting with the first test case. The conclusion is also accepted (the "True Result" column of Table 5.3 illustrates that the conclusion, fact 2, is accepted) by the interpreter so the example story problem is successfully solved.

Even though a case is found where both the reasons and the fact evaluate to true, the rest of the cases have to be checked. The fact must evaluate to true for every test case in which all the reasons evaluate to true. The fact is rejected, if a case is found where all reasons evaluate to true and the fact evaluates to false.

If all test cases produce a false result for at least one of the reasons or the fact never produces a true result, meaning the TrueCount remains zero, then the fact is not accepted.

If a fact is accepted, then that fact's pcodes are added to the fact-array. Once added to the fact-array, the fact is considered true and can be used as a reason for future facts. For example in the sample problem, the first fact is accepted and then it is used as a reason for the conclusion.

Once all facts ending with the conclusion are accepted, the execution ends and the program, the list of facts with reasons, is a correct solution to the story problem.

A user is not limited on the number of reasons given for the facts for each step in the program; nor is there a limited on the number of actual steps for the program.

By examining Table 5.3, if a user gives one fact, which would be the conclusion, in the program for the story problem, depending on the reasons used, the interpreter returns two conflicting results. If a user gives axiom 1 and axiom 2 as the reasons, in order to be accepted, the conclusion has to be true for test cases one, five, seven, and eight because those are the cases in which both reasons are true. Since the conclusion is not true for case seven or eight, the interpreter rejects it. On the other hand, if a user uses axiom 1 and 2 and premise 1 as reasons the interpreter accepts the conclusion because all three reasons are true for only the first test case and the conclusion is also true for this case.

Chapter 6

PREDICATE LANGUAGE

The internal language structure -- the pcodes

The "pcodes" for the predicate interpreter are listed in the third table in Appendix D: "Predicate Opcode Table". The first pcode is "olang 3 0" and the last one is "oend 0 0".

An example story problem and program are listed in Figure 6.1 and the corresponding pcodes are illustrated in Table 6.1. The operands "jinx" and "kinx" listed in Table 6.1, are indices into the name table. They map to j and k which were used as range bounds in the story problem. The "binx" operand represents the b array and "iinx" is the bound variable used to index the array. At run time, the operands for "oeq" will be at the top of the stack; therefore, "0" is used in the pcode. The first operand of the operator "opushi" (push immediate) is "0" which is the "0" used in the program.

```
Story Problem:
     ¬some values of b[j..k] are zero
Program:
    ( ThereExists i : (j..k) : b[i] = 0 )
```

Figure 6.1 : Example predicate story problem and program

Table 6.1 : Predicate sample pcodes

Operator	Operand	Operand
olang	3	0
oexist	iinx	0
orange	jinx	kinx
oindex	binx	iinx
opushi	0	0
oeq	0	0
oend	0	0

The test case file

In testing a program in the predicate problem domain, if the number of test cases is determined as it is in the other two domains, then the test case file would be very large. The example: (ThereExists i : (j..k) : b[i] = 0) is equivalent to (b[j]=0 or b[j+1]=0 or ... or b[k-1]=0 or b[k]=0) which yields two raised to the power of "k-j+1" test cases. That many test cases is not practical so a heuristical approach is taken which uses a small number of test cases. The actual number of test cases is a judgement call, it could be two, three, four, or any number decided by the program solution writer. For the example illustrated in this chapter, two test cases are used.

The beginning of the test case file has values for each element of the program solution array, these array values are used for each of the test cases.

Each test case is comprised of a value for the lower bound variable of the range and one for the upper bound variable. The predicate is executed once for each value of each test range.

To derive the test case values, create an array and a table similar to the ones in Figure 6.2. The size of the array is arbitrary; however, it depends somewhat on the values that fill it. For the example in Figure 6.1, an array of ten elements was used and the array values are displayed in Figure 6.2b. Only array elements 3, 4, and 6 satisfy the predicate; therefore, at least one of the test case ranges must include at least one of these elements and at least one of the test case ranges must exclude all of these elements.

a. Predicate :

```
( ThereExists i : ( j .. k ) : b[i] = 0 )
```

b. Array values :

```
b[1] = 3    b[2] = 6    b[3] = 0    b[4] = 0    b[5] = 4
b[6] = 0    b[7] = 6    b[8] = 5    b[9] = 3    b[10] = 7
```

c. Test case table :

Case Number	Lower Bound	Upper Bound	Test Case Value	Counting Variable Result
1	2	5	true	2
2	7	10	false	0

Figure 6.2 : Array & table used to derive example predicate test case file

When the test cases are created for the predicate problem domain, the story problem needs to be examined carefully. A user is not restricted to writing an exact duplicate of the solution program for a particular story problem; therefore, the test cases should be created so that "almost correct" programs will not be accepted.

For the example predicate, the following incorrect program is accepted by the interpreter because the first test case (see Figure 6.2c) has two array elements equal to zero in its range.

```
( NumberOf i : ( j..k ) : ( b[i] = 0 ) > 1 ).
```

This predicate actually means "in the range there is more than one zero"; however, the story problem stated "at least one zero exists". To correct the first test case so that the above program is not accepted the range should have only one array element equal to zero, not two.

The test case file structure

Each value in the predicate problem domain test case file is on a line by itself beginning in column one and no blank lines exist within the file. The first part of the test case file has a value for each element in the array. The array element values are listed sequentially starting with the first element. The array name is listed before each array element value.

The rest of the test case file is comprised of the actual test cases. Each test case contains values for the lower and upper range variables, and a boolean value for the test case. The name of each range variable is listed before its value and the symbol "&" is listed before the boolean value. The boolean value is true if the test case range satisfies the predicate; it is false otherwise. True is represented by "1" and false by "0".

Figure 6.3 illustrates the test case file for the example problem described in Figure 6.1. The "b", (Figure 6.3) is the name of the array, which has ten elements, indexed one through ten. The "j" is the lower bound variable and "k" is the upper bound. Each test case uses the same values for the array.

```
b CR   3 CR     b CR   6 CR     b CR   0 CR
b CR   0 CR     b CR   4 CR     b CR   0 CR
b CR   6 CR     b CR   5 CR     b CR   3 CR
b CR   7 CR
j CR   2 CR     k CR   5 CR     & CR   1 CR
j CR   7 CR     k CR  10 CR     & CR   0 CR
```

Figure 6.3 : Predicate test case file example

The interpreter design

The pcodes for the predicate problem domain contains several parts: a language number, a qualifier, two range variables, and the proposition. The predicate interpreter executes the pcodes for the proposition part of the predicate using the range values from each test case and evaluates the true/false result for each test case. For the user's program to be a correct interpretation of the story problem, when the program is executed with each test case range, the result must match the given test case answer, the value after the "&" token in the test case file.

At the start of each test case, a counting variable is set to zero. Each time the execution of the pcodes for the proposition yields "true", the counting variable is incremented. After the last execution of the proposition pcodes, the counting variable is evaluated to determine if the entire test case yields "true" or "false".

The counting variable value varies depending on which quantifier is used. For a program which used "ThereExists", if the counting variable is greater than zero then the test case result is true, else it is false. For "ForAll", if the counting variable is equal to the upper bound minus the lower bound plus one then the result is true, else false. For "NumberOf", if the counting variable is equal to the value represented by the first operand in the pcode for the operator "onum" then the result is true, else false.

If the test case result matches the value in the test case file, the value after the "&" token, then the next test case is processed. If the result does not match then the interpreter halts execution and returns the message "your program is not correct!"; if each test case matches the test case value then the message "Your program is correct!" is returned to the user.

Chapter 7

CONCLUSION

Code-generator/grammar problems

During the design and implementation of the three interpreters, three problems related to the grammars and code generators were encountered. The next few paragraphs discuss these problems and their solutions.

Each problem domain's grammar allows relational expressions to be structured like the examples in Figure 7.1. Figure 7.2 illustrates the pcodes generated for each example in Figure 7.1.

a. x > y > z **b.** s < t < s **c.** p > q = r

Figure 7.1 : Relational expression examples

```
a.  opush xinx 0    b.  opush sinx 0    c.  opush pinx 0
    opush yinx 0        opush tinx 0        opush qinx 0
    ogrt  0    0        olss  0    0        ogrt  0    0
    opush zinx 0        opush sinx 0        opush rinx 0
    ogrt  0    0        olss  0    0        oeq   0    0
```

Figure 7.2 : Pcodes for relational expression examples

When either of the three interpreters executes the pcodes in Figure 7.2, even though the original expressions were correct, the interpreters will reject them. For example, after executing the pcodes in Figure 7.2c when the values 3, 2, 1 are used for p, q, r respectively, each interpreter returns "true"; however, the actual result is "false".

Each interpreter uses "1" to represent "true" and "0" for "false". When executing the pcodes for Figure 7.2c (again using the test values 3, 2, 1 for p, q, r) an interpreter pushes the value for p onto the run stack and then pushes the value of q. The next pcode requires an interpreter to pop two values off the run stack and perform the "greater than" operation which, in turn, pushes the boolean value "1" onto the stack. The next pcode pushes the value for r onto the stack and the "oeq" pcode performs the "equals" operation. The operands for the "oeq" pcode are 1 (true) and 1 (value of r) when they should have been 2 (value of q) and 1 (value of r).

The solution would require the code generators to optimize the pcodes when relational expressions are structured as those illustrated in Figure 7.1. For example, Figure 7.2c would have the pcode "opush qinx 0" after the pcode "ogrt 0 0". Then each interpreter would compare the value for "q" with the value for "r" verses comparing a boolean value with the value for "r".

The second problem deals with logical expressions. The source grammars allow logical expressions in the form illustrated by the examples of Figure 7.3. The precedence rules for "and", "or", "implies", and "equivalence" are ignored in the source grammars; however, "not" is handled correctly. Figure 7.4 demonstrates the pcodes for the examples in Figure 7.3. The pcodes illustrate that the equation will be evaluated left to right irregardless of the precedence. Without precedence rules, Figure 7.3a would be a different expression if it were written "h and g or f".

a.f or g and h **b.** l implies m equals n **c.** i or j implies k

Figure 7.3 : Logical expression examples

```
a. opush finx 0     b. opush linx 0     c. opush iinx 0
   opush ginx 0        opush minx 0        opush jinx 0
   oor   0    0        oimp  0    0        oor   0    0
   opush ginx 0        opush ninx 0        opush kinx 0
   oand  0    0        oeqv  0    0        oimp  0    0
```

Figure 7.4 : Pcodes for logical expression examples

The solution would leave all parts of the system unchanged, which means that all logical operators have equal precedence. (Perhaps the user would be warned that the logical operators have equal precedence and that by default, the answer is executed left to right.) The user can enforce the order of execution, however, with parentheses i.e. "f or (g and h)" verses "f or g and h". or with reordering the expressions, i.e. "g and h or f" verses "f or g and h".

The third problem is with the predicate language because it has trouble dealing with the operator "not". The grammar allows the "not" to be taken on the entire equation; however, this cannot be done by the interpreter because it takes the "not" of the current value on the stack. The problem can be illustrated with the example predicate and pcodes in Figure 7.5. The interpreter takes the "not" of the value placed on the stack by the "oeq" pcode instead of executing the "not" for the entire predicate.

```
Predicate :
  not ( ForAll i : ( j..k ) : b[i] = 0 )

Pcode : olang  3    0
        oall   iinx 0
        opush  iinx 0
        opush  iinx 0
        orange 0    0
        opush  ainx 0
        opush  iinx 0
        oindex 0    0
        opushi 0    0
        oeq    0    0
        onot   0    0
        oend   0    0
```

Figure 7.5 : Predicate and pcodes example

The solution would append "oend" to the end of the rule "Lexp". (Refer to the Predicate translation grammar in Appendix B for the "Lexp" rule.) With the change, the code generator inserts the pcode "oend 0 0" before the pcode "onot 0 0", which causes the interpreter to first execute the proposition part of the predicate once for each value of the range, then take the "not" of the evaluation of the counting variable value.

Testing of the interpreters

The proposition and predicate interpreters were tested with a number of different story problems and sample programs. These two interpreters include a procedure which allows the stack changes to appear on the screen. From viewing the stack operations, the proposition and predicate interpreters work as expected from their design.

The design of the proposition proof interpreter required that some of the pcodes be saved at the start of a proof program execution and that during the execution more pcodes would also be saved. Then the execution of a proof program would either require the user to give all facts and reasons at once or would require the interpreter to execute each fact as a separate program. The second method would mean that those pcodes which need to be saved would be saved in a file. This design issue was never resolved; therefore, the interpreter was never completed.

REFERENCES

Barrett, William A., Bates, Rodney M., Gustafson, David A., Couch, John D. Compiler Construction Theory And Practice, Chicago, Illinois: Science Research Associates, Inc., 1986. Chapter 2, p. 17-66.

Bauer, F. L., Eickel, J. Compiler Construction an Advanced Course. New York: Springer-Verlag, 1976. Chapter 1, p. 1-36, Chapter 2, p. 37-56.

Borland International, TurboPascal Version 3.0 Reference Manual, California, 1985.

Gries, David. The Science of Programming. New York: Springer-Verlag New York Inc., 1981. pp. 8-9,12, 25, 29, 66, 71-74, 304--309.

Korfhage, Robert R. Logic and Algorithms: with Applications to the Computer and Information Sciences. New York: John Wiley & Sons Inc., 1966. Chapter 6, p. 125-143.

Pratt, Terrence W. Programming Languages Design and Implementation. New Jersey: Prentice-Hall, 1984. Chapter 9, p. 303-329.

APPENDIX A

Proposition Source Grammar

```
Props ::= [ :Lab: Prop eol ]*  Prop eol
      {*"Lab" defines a label, if the line is correct*}

Prop  ::= Lexp [ Lop Lexp ]*

Lexp  ::=   Lab | Rexp | not Lexp | ( Prop )
          | Pvar ( Ovar )
      {*Note: label must have been previously defined*}

Lop   ::= and | or | imp | eqv

Rexp  ::= Exp [ Relop Exp ]1..2 | Ivar in Range

Relop ::=  = | =< | >= | /= | > | <

Range ::= ( Iexp .. Iexp )

Exp   ::= Var | Avar "[" Iexp "]" | Iexp

Iexp  ::= Const | Ivar [ Sign Const ]

Var   ::= { any "var" name defined in the problem }

Avar  ::= { any "array" name defined in the problem }

Ivar ::={ any "index" name, defined in
           the problemorin Qualex or Numex }

Pvar  ::= { any "pred" name defined in the problem }

Cvar  ::= { any "const" name defined in the problem }

Ovar  ::= { any "object" name defined in the problem }

Lab   ::= { any "label" name }

     {*Note: problem names are those
      used in the story problem*}

Sign  ::= + | -

Const ::= Int | Cvar

Int   ::= {any legal integer}
```

Proposition Proof Source Grammar

```
Prog   ::= Step eol [ Step eol ]* done.

Step   ::= :"F"Int : Lexp [ Lop Lexp ]* // Reasons

Lexp   ::=  Lab | Rexp | not Lexp | ( Prop )
          | Pvar ( Ovar )
       {*Note: label must have been previously defined*}

Lop    ::= and | or | imp | eqv

Rexp   ::= Exp [ Relop Exp ]1..2 | Ivar in Range

Relop ::=  = | =< | >= | /= | > | <

Range ::= ( Iexp .. Iexp )

Exp    ::= Var | Avar "[" Iexp "]" | Iexp

Iexp   ::= Const | Ivar [ Sign Const ]

Var    ::= { any "var" name defined in the problem }

Avar   ::= { any "array" name defined in the problem }

Ivar ::={ any "index" name, defined in
           the problemorin Qualex or Numex }

Pvar   ::= { any "pred" name defined in the problem }

Cvar   ::= { any "const" name defined in the problem }

Ovar   ::= { any "object" name defined in the problem }

Lab    ::= { any "label" name }

      {*Note: problem names are those
        used in the story problem*}

Sign   ::= + | -

Const ::= Int | Cvar

Int    ::= {any legal integer}

Reasons ::= Reason [ , Reason ]*

Reason  ::= "A"Int | "P"Int | "F"Int
```

Predicate Source Grammar

```
Prog    ::= [ :Lab: Prop eol ]* Pred
          {*"Lab" defines a label, if the line is correct*}

Pred    ::= Qualex | Numex Relop Iexp

Qualex ::= [ not ] ( Qual Ivar : Range : Lexp )

Qual    ::= "All" | "Exist"

Numex   ::= ( "Num" Ivar : Range : Lexp )

Lexp    ::=   Lab | Rexp | not Lexp | ( Prop )
            | Pvar ( Ovar )
          {*Note: label must have been previously defined*}

Lop     ::= and | or | imp | eqv

Rexp    ::= Exp [ Relop Exp ]1..2 | Ivar in Range

Relop   ::=  = | =< | >= | /= | > | <

Range   ::= ( Iexp .. Iexp )

Exp     ::= Var | Avar "[" Iexp "]" | Iexp

Iexp    ::= Const | Ivar [ Sign Const ]

Var     ::= { any "var" name defined in the problem }

Avar    ::= { any "array" name defined in the problem }

Ivar   ::={ any "index" name, defined in
             the problemorin Qualex or Numex }

Pvar    ::= { any "pred" name defined in the problem }

Cvar    ::= { any "const" name defined in the problem }

Ovar    ::= { any "object" name defined in the problem }

Lab     ::= { any "label" name }
        {*Note: problem names are those
          used in the story problem*}

Sign    ::= + | -

Const   ::= Int | Cvar

Int     ::= {any legal integer}
```

Notation used :

1. Each entry in the source grammars is a rule. The name of a rule is listed on the left side of the symbols: "::=". The choices for a rule are listed after the "::=" symbols and are separated with the symbol "|".

2. The square brackets "[" and "]" enclose items that are optional.

   a. The asterisk means zero or more occurrences.

   b. The "1..2" means one or two occurrences.

   c. No symbol means zero or one occurrence.

3. To distinguish the difference between the use of the square brackets for optional items and the use of them as part of a choice for a rule, they are enclosed in quotes when they are part of a choice. For example, the square brackets in the second choice of the rule "exp" are required as part of that choice so they are enclosed in quotes.

4. Quotes are also used around words when the actual word is part of a choice; for example, the "Qual" rule is either the actual word "All" or the actual word "Exists".

5. The symbols "{*" and "*}" are used for comments about the rule, such as after the rules Props, Lexp, and Lab in the proposition source grammar.

6. The curly brackets "{" and "}" are used to list a class of items such as the choices for the rules Var, Avar, and Ivar in the proposition source grammar.

APPENDIX B

Proposition Translation Grammar

```
Props ::= { "olang 1" }
          [ :Lab: Prop eol { define(Lab) } ]*  Prop
          { "oend" }

Prop  ::= Lexp [ Lop Lexp { olop } ]*

Lexp  ::=  Lab  { output code generated
                  by previous labeled line }
         | Rexp
         | not Lexp { "onot" }
         | ( Prop )
         | Pvar ( Ovar ) { "oarg oinx" "ofun pinx" }
        {*Note: "oinx" in the Ovar index,
                "pinx" is the Pvar index*}

Lop   ::=  and { olop <-- "oand" }
         | or  { olop <-- "oor"  }
         | imp { olop <-- "oimp" }
         | eqv { olop <-- "oeqv" }

Rexp  ::=  Exp  [ Relop Exp { rel } ]
           [ Relop Exp { rel } ]
         | Ivar in Range { "oin iinx" }
        {*Note: "iinx" is the Ivar index*}

Relop ::=  =  { rel <-- "oeq"  }
         | =< { rel <-- "oels" }
         | >= { rel <-- "ogeq" }
         | /= { rel <-- "oneq" }
         | >  { rel <-- "ogrt" }
         | <  { rel <-- "olss" }

Range ::= ( Iexp .. Iexp ) { "orange" }

Exp   ::=  Var { "opush inx" }
         | Avar { "opush ainx" } "[" Iexp "]"
           { "oindex" }
         | Iexp
        {*Note: "inx" is the Var index,
                "ainx" is the Avar index*}

Iexp  ::=  Const
         | Ivar { opush iinx" } [ Sign Const { sign } ]

Sign  ::=  + { sign <-- "oplus" }
         | - { sign <-- "ominus" }
```

```
Const ::=  Int { "opushi int" }
         | Cvar { "opush cinx" }
        {*Note: "cinx" is the Cvar index*}
```

Proposition Proof Translation Grammar

```
Prog    ::= { olang 2" } [ Step eol ]* Last { "oend" }

Step    ::= :"F"Int : { "ofact finx" }
              Lexp [ Lop Lexp ]* // { "owhy" } Reasons
            {*Note: "finx" is the fact index*}

Last    ::= :"C": { "ofact cinx" }
              // { "owhy" } Reasons
            {*Note: "cinx" is the conclusion index*}

Lexp   ::=  Lab  { output code generated
                   by previous labeled line }
          | Rexp
          | not Lexp { "onot" }
          | ( Prop )
          | Pvar ( Ovar ) { "oarg oinx" "ofun pinx" }
        {*Note: "oinx" in the Ovar index,
               "pinx" is the Pvar index*}

Lop    ::=  and { olop <-- "oand" }
          | or  { olop <-- "oor"  }
          | imp { olop <-- "oimp" }
          | eqv { olop <-- "oeqv" }

Rexp   ::=  Exp  [ Relop Exp { rel } ]
            [ Relop Exp { rel } ]
          | Ivar in Range { "oin iinx" }
        {*Note: "iinx" is the Ivar index*}

Relop ::=  =  { rel <-- "oeq"  }
         | =< { rel <-- "oels" }
         | >= { rel <-- "ogeq" }
         | /= { rel <-- "oneq" }
         | >_ { rel <-- "ogrt" }
         | <  { rel <-- "olss" }

Range ::= ( Iexp .. Iexp ) { "orange" }

Exp    ::=  Var { "opush inx" }
          | Avar { "opush ainx" } "[" Iexp "]"
            { "oindex" }
          | Iexp
        {*Note: "inx" is the Var index,
               "ainx" is the Avar index*}
```

```
Iexp   ::=  Const
          | Ivar { opush iinx" } [ Sign Const { sign } ]

Sign   ::=  + { sign <-- "oplus" }
          | - { sign <-- "ominus" }

Const ::=  Int { "opushi int" }
         | Cvar { "opush cinx" }
       {*Note: "cinx" is the Cvar index*}

Reasons ::= Reason [ , Reason ]* { "oend" }

Reason  ::=    "A"Int { "oaxiom ainx" }
             | "P"Int { "oprem pinx"  }
             | "F"Int { "ofact finx"  }
             { note: "ainx" is the axiom index,
                     "pinx" is the premise index }
```

Predicate Translation Grammar

```
Prog    ::= { "olang 3" }
            [ :Lab: Prop ]* Pred { "oend" }

Pred    ::= Qualex | Numex Relop Iexp { rel }

Qualex ::= { not <-- null } [ not { not <-- "onot" } ]
            ( Qual Ivar { qual "iinx" } : Range : Lexp )
            { not }

Qual    ::=    "All"   { qual <-- "oall"   }
             | "Exist" { qual <-- "oexist" }

Numex   ::= ( "Num" Ivar { "onum iinx " } : Range :
            Lexp { "oend" } )

Lexp   ::=  Lab  { output code generated
                   by previous labeled line }
          | Rexp
          | not Lexp { "onot" }
          | ( Prop )
          | Pvar ( Ovar ) { "oarg oinx" "ofun pinx" }
        {*Note: "oinx" in the Ovar index,
                "pinx" is the Pvar index*}
Lop    ::=  and { olop <-- "oand" }
          | or  { olop <-- "oor"  }
          | imp { olop <-- "oimp" }
          | eqv { olop <-- "oeqv" }
```

```
Rexp   ::=  Exp  [ Relop Exp { rel } ]
            [ Relop Exp { rel } ]
          | Ivar in Range { "oin iinx" }
        {*Note: "iinx" is the Ivar index*}

Relop ::=   =  { rel <-- "oeq"  }
          | =< { rel <-- "oels" }
          | >= { rel <-- "ogeq" }
          | /= { rel <-- "oneq" }
          | >  { rel <-- "ogrt" }
          | <  { rel <-- "olss" }

Range ::= ( Iexp .. Iexp ) { "orange" }

Exp    ::=  Var { "opush inx" }
          | Avar { "opush ainx" } "[" Iexp "]"
            { "oindex" }
          | Iexp
        {*Note: "inx" is the Var index,
                "ainx" is the Avar index*}

Iexp   ::=  Const
          | Ivar { opush iinx" } [ Sign Const { sign } ]

Sign   ::=  + { sign <-- "oplus" }
          | - { sign <-- "ominus" }

Const ::=   Int { "opushi int" }
          | Cvar { "opush cinx" }
        {*Note: "cinx" is the Cvar index*}
```

Notation used :

1. Each entry in the translation grammars is a rule. The name of a rule is listed on the left side of the symbols: "::=". The choices for a rule are listed after the "::=" symbols and are separated with the symbol "|".

2. The square brackets "[" and "]" enclose items that are optional.

   a. The asterisk means zero or more occurrences.

   b. No symbol means zero or one occurrence.

3. To distinguish the difference between the use of the square brackets for optional items and the use of them as part of a choice for a rule, they are enclosed in quotes when they are part of a choice. For example, the square brackets in the second choice of the rule "exp"

are required as part of that choice so they are enclosed in quotes.

4. Quotes are also used around words when the actual word is part of a choice; for example, the "Qual" rule is either the actual word "All" or the actual word "Exists".

5. The symbols "{*" and "*}" are used for comments about the rule, such as after the rules Props, Lexp, and Lab in the proposition source grammar.

6. Some of the items inside the curly brackets are intermediate steps towards the pcodes. An example of an intermediate step is shown in the "Prog" rule of the predicate translator grammar. The { "olang 3" } is added to the front of the "Prog" rule of the predicate source grammar and { "oend" } is added to the end which creates the "Prog" rule of the predicate translation grammar. The intermediate steps are replaced by terminals, which are partial pcodes, in the respective target grammar.

7. The rest of the items in the curly brackets are semantic actions. Semantic actions are distinguished from the intermediate steps by the use of the symbols "<--" which means replace the left side with the value on the right side. The replace action is the semantic action. For example, in the "Relop" rule, the first choice is to replace the "rel" with the operator "oeq". The replacement takes place in the rule "Rexp" because the term "rel" is located in "Rexp"; the semantic action, however, is described in the "Relop" rule.

APPENDIX C

Proposition Target Grammar

```
Props  ::= "olang" "1" [ Lab Prop ]* Prop "oend"

Prop   ::= Lexp [ Lexp Lop ]*

Lexp   ::=
          | Rexp
          | Lexp "onot"
          | Prop
          | "oarg" "oinx" "ofun" "pinx"

Lop    ::= "oand" | "oor" | "oimp" | "oeqv"

Rexp   ::= Exp [ Exp Relop ] [ Exp Relop ]

Relop  ::=  "oeq"   | "oels"
          | "ogeq"  | "oneq"
          | "ogrt"  | "olss"

Range  ::= Iexp Iexp "orange"

Exp    ::=  "opush" "inx"
          | "opush" "ainx" Iexp "oindex"
          | Iexp

Iexp   ::=  Const
          | "opush" "iinx" [ Sign Const ]

Sign   ::= "oplus" | "ominus"
```

Proposition Proof Target Grammar

```
Prog    ::=  "olang" "2" [ Step ]* Last "oend"

Step    ::= "ofact" "finx" Lexp [ Lop Lexp ]*
            // "owhy" Reasons

Last    ::= "ofact" "cinx"
            // "owhy" Reasons

Lexp   ::=
          | Rexp
          | Lexp "onot"
          | Prop
          | "oarg" "oinx" "ofun" "pinx"
```

```
Lop    ::= "oand" | "oor" | "oimp" | "oeqv"

Rexp   ::= Exp [ Exp Relop ] [ Exp Relop ]

Relop ::=   "oeq"   | "oels"
          | "ogeq"  | "oneq"
          | "ogrt"  | "olss"

Range ::= Iexp Iexp "orange"

Exp    ::=   "opush" "inx"
          | "opush" "ainx" Iexp "oindex"
          | Iexp

Iexp   ::=   Const
          | "opush" "iinx" [ Sign Const ]

Sign   ::= "oplus" | "ominus"

Reasons ::= Reason [ Reason ]* "oend"

Reason   ::=  "oaxiom" "ainx"
            | "oprem" "pinx"
            | "ofact" "finx"
```

Predicate Target Grammar

```
Prog    ::= "olang" "3" [ Lab Prop ]* Pred "oend"

Pred    ::= Qualex | Numex Iexp Relop

Qualex ::= Qual "iinx" Range Lexp [ "onot" ]

Qual    ::= "oall" | "oexist"

Numex   ::= "onum" "iinx" Range Lexp

Lexp   ::=
          | Rexp
          | Lexp "onot"
          | Prop
          | "oarg" "oinx" "ofun" "pinx"

Lop    ::= "oand" | "oor" | "oimp" | "oeqv"
Rexp   ::= Exp [ Exp Relop ] [ Exp Relop ]

Relop ::=   "oeq"   | "oels"
          | "ogeq"  | "oneq"
          | "ogrt"  | "olss"
```

```
Range ::= Iexp Iexp "orange"

Exp   ::=   "opush" "inx"
          | "opush" "ainx" Iexp "oindex"
          | Iexp

Iexp  ::=   Const
          | "opush" "iinx" [ Sign Const ]

Sign  ::= "oplus" | "ominus"
```

Notation used :

1. Each entry in the target grammars is a rule. The name of a rule is listed on the left side of the symbols: "::=". The choices for a rule are listed after the "::=" symbols and are separated with the symbol "|".

2. The square brackets "[" and "]" enclose items that are optional.

   a. The asterisk means zero or more occurrences.

   b. No symbol means zero or one occurrence.

3. The items enclosed in quotes are the pcode operators and operands. These quotes are not part of the pcodes; they are used to make the operators and operands easier to identify.

4. The operands which end in "inx" are actually indices into the name table. For example, the "ainx" operand from the "Exp" rule is the index value of the name table element where the array information is located.

APPENDIX D

Proposition Language Opcode Table

Operator	Meaning	Interpreter Code
oeq	equals	push (right (Operand2) = left (Operand1))
oneq	not equals	push (right (Operand2) <> left (Operand1))
olss	less than	push (right (Operand2) > left (Operand1))
oels	less than or equal	push (right (Operand2) >= left (Operand1))
ogrt	greater than	push (right (Operand2) < left (Operand1))
ogeq	greater than or equal	push (right (Operand2) <= left (Operand1))
oand	logical and	push ((right (Operand2) = aTrue) and (left (Operand1) = aTrue))
oor	logical or	push ((right (Operand2) = aTrue) or (left (Operand1) = aTrue))
oimp	logicalimplies	If (right(Operand2) = aFalse) or (left (Operand1) = aTrue) Then push(True) Else push(False) {endif}
oeqv	logical equivalence	If right (Operand2) = left (Operand1) Then push (True) Else push (False); {endif}

opush	push	pushint(left(Operandl));
opushi	push immediate	pushint(Operandl);
olang	language	If not Operandl = 1 Then error := True; {endif}
oend	end terminator	Case 1: Used to mark the end of the "owhy" pcode; Case 2: Used to mark the end of the pcode.
oarg	argument	;
ofun	function	pushint (left (Operandl));
oplus	add Operandl to Operand2	push (right (Operand2) + left (Operandl))
ominus	subtract Operandl from Operand2	push (right (Operand2) - left (Operandl))
ostar	multiply Operandl and Operand2	push (right (Operand2) * left (Operandl))
odiv	divide Operand2 by Operandl	push (right (Operand2) / left (Operandl))
onot	not	pop (bool); If bool = 1 Then bool := 0 Else bool := 1 {endif} push(bool);

Proposition Proof Language Opcode Table

Operator	Meaning	Interpreter Code
oaxiom	axiom	Executes pcode listed in labels array for the axiom identified by Operandl.

oprem	premise	Executes pcode listed in labels array for the premise identified by Operand1.
ofact	fact	Case 1: Uses it as the beginning marker to the pcode for the fact; Case 2: Executes pcode listed in labels array for the fact identified by Operand1.
owhy	why - reason	Uses it as the beginning marker to the pcode which lists the reasons the fact is true, pcode that would follow owhy would be oaxiom operand1 Operand2, oprem Operand1 Operand2, or ofact Operand1 Operand2.
oeq	equals	push (right (Operand2) = left (Operand1))
oneq	not equals	push (right (Operand2) <> left (Operand1))
olss	less than	push (right (Operand2) > left (Operand1))
oels	less than or equal	push (right (Operand2) >= left (Operand1))
ogrt	greater than	push (right (Operand2) < left (Operand1))
ogeq	greater than or equal	push (right (Operand2) <= left (Operand1))
oand	logical and	push ((right (Operand2) = aTrue) and (left (Operand1) = aTrue))
oor	logical or	push ((right (Operand2) = aTrue) or (left (Operand1) = aTrue))

```
oimp      logicalimplies          If (right(Operand2) =
                                               aFalse) or
                                     (left (Operand1)  =
                                                aTrue) Then
                                     push(True)
                                  Else
                                     push(False)
                                  {endif}

oeqv      logical equivalence     If right (Operand2) =
                                     left  (Operand1) Then
                                     push (True)
                                  Else
                                     push (False);
                                  {endif}

opush     push                    pushint(left(Operand1));

opushi    push immediate          pushint(Operand1);

olang     language                If not Operand1 = 1 Then
                                     error := True;
                                  {endif}

oend      end terminator          Case 1: Used to mark the
                                  end of the "owhy" pcode;
                                  Case 2: Used to mark the
                                  end of the pcode.

oarg      argument                ;

ofun      function                pushint (left (Operand1));

oplus     add Operand1            push (right (Operand2) +
          to Operand2                      left  (Operand1))

ominus    subtract Operand1       push (right (Operand2) -
          from Operand2                    left  (Operand1))

ostar     multiply Operand1       push (right (Operand2) *
          and Operand2                     left  (Operand1))

odiv      divide Operand2         push (right (Operand2) /
          by Operand1                      left  (Operand1))
```

onot	not	pop (bool);
		If bool = 1 Then
		   bool := 0
		Else
		   bool := 1
		{endif}
		push(bool);

Predicate Language Opcode Table

Operator	Meaning	Interpreter Code

oexist	there exists	If trueCount = 0 Then
		   error := True;
		{endif}

oall	for all	If trueCount <>
		   highInx-lowInx+1 Then
		   error := True;
		{endif}

onum	number	If trueCount <>
		   {onum value} Then
		   error := True;
		{endif}

oindex	index	pushint(B[boundVar]);

orange	range	Begin
		{InterpTab should have const values in the field "value" for jinx & kinx which was set by calling addinx}
		   If nd2 <> 0 Then
		      highInx := nd2
		   Else
		      error := True;
		   {endif}
		   If nd1 <> 0 Then
		      lowInx := nd1
		   Else
		      error := True
		   {endif}
		End;

oin	in - used with range	

```
oplus      add Operand1               push (right (Operand2) +
           to Operand2                      left  (Operand1))

ominus     subtract Operand1          push (right (Operand2) -
           from Operand2                    left  (Operand1))

ostar      multiply Operand1          push (right (Operand2) *
           and Operand2                     left  (Operand1))

odiv       divide Operand2            push (right (Operand2) /
           by Operand1                      left  (Operand1))

onot       not                        pop (bool);
                                      If bool = 1 Then
                                         bool := 0
                                      Else
                                         bool := 1
                                      {endif}
                                      push(bool);

oeq        equals                     push (right (Operand2) =
                                            left  (Operand1))

oneq       not equals                 push (right (Operand2) <>
                                            left  (Operand1))

olss       less than                  push (right (Operand2) >
                                            left  (Operand1))

oels       less than or equal         push (right (Operand2) >=
                                            left  (Operand1))

ogrt       greater than               push (right (Operand2) <
                                            left  (Operand1))

ogeq       greater than or equal      push (right (Operand2) <=
                                            left  (Operand1))

oand       logical and                push ((right (Operand2) =
                                                      aTrue) and
                                             (left  (Operand1)  =
                                                      aTrue))

oor        logical or                 push ((right (Operand2) =
                                                      aTrue) or
                                             (left  (Operand1)  =
                                                      aTrue))
```

oimp	logicalimplies	If (right(Operand2) = aFalse) or (left (Operand1) = aTrue) Then push(True) Else push(False) {endif}
oeqv	logical equivalence	If right (Operand2) = left (Operand1) Then push (True) Else push (False); {endif}
opush	push	pushint(left(Operand1));
opushi	push immediate	pushint(Operand1);
olang	language	If not Operand1 = 1 Then error := True; {endif}
oend	end terminator	Case 1: Used to mark the end of the "owhy" pcode; Case 2: Used to mark the end of the pcode.
oarg	argument	;
ofun	function	pushint (left (Operand1));

Notes:

1. From the pcode format, operator operand operand, the first operand is referred to as "Operand1" in this appendix and the second one is referred to as "Operand2".

2. The terms "push" and "pushint" from the "Interpreter Code" column are function calls. "Push" puts the value on the top of the run stack. "Pushint" puts Operand1 on the top of the run stack.
3. The terms "left" and "right" from the "Interpreter Code" column are functions calls. The functions either retrieve the values from the name table or pop them off the top of the run stack.

4. The "trueCount" used in the "Interpreter Code" column is set to zero at the start of the interpreter. Each time the pcode is executed and the result is true, trueCount is incremented.

5. The variable "highInx" from the "Interpreter Code" column holds the value of the upper bound of the range and the variable "lowInx" holds the value of the lower bound of the range.

APPENDIX E

*** This is the code for the file "Intrl.inc" ***

```
{general interpreter symbol table definitions }
   Const
      InterpSize = 400;
   Type
      InterpRecord = Record
                        name :  nametype;
                        kind :  char;
                        value : Integer
                     End;
      InterpTabType = Array [1..InterpSize] of InterpRecord;
   Var
      InterpTab    : InterpTabType;

Procedure initInterpTab;
   Var
      inx : Integer;
      txt : string[255];
   Begin
      For inx := 1 to InterpSize Do
      Begin
         InterpTab[inx].name := '          ';
         InterpTab[inx].kind := 'o';
         InterpTab[inx].value := -1;
      End; {for}
   End; {initInterpTab}

Procedure addindex (name : nametype;
                    kind : char;
                    inx  : Integer);
   Begin
      InterpTab[inx].name := name;
      InterpTab[inx].kind := kind;
   End; {addindex}
```

*** This is the code for the file "Intr2.inc" ***

```
Const
   MaxStackSize = 25;
   aTrue = 1;
   aFalse = 0;

Type
   StackType = Array [1..MaxStackSize] of Integer;
```

```
Var
   answer : Char;
   B : Array [1..10] of Integer;
   Stack : StackType;
   name : nametype;
   inx,
   jnx,
   whichnum,
   savefact,
   savereasons,
   work,
   value,
   newVal,
   trueCount,
   StartOfCode,
   boundVar,
   low,
   high,
   highInx,
   lowInx,
   index,
   StackIndex : Integer;
   theCase : optype;
   showstack : Boolean;

{InitStack initializes the run stack to a default value}
Procedure InitStack;

   Var
      inx : Integer;

   Begin
      For inx := 1 to MaxStackSize Do
         Stack[inx] := maxint
      {endfor}
   End; {InitStack}
```

```
{StoreVal stores "value" into the InterpTab at the location}
{where "name" is                                            }
Procedure StoreVal (name  : nametype;
                    value : Integer);

   Var
      index : Integer;

   Begin
      index:=1;
      While index <= InterpSize Do
      Begin
         If InterpTab[index].name = name Then
         Begin
            InterpTab[index].value := value;
            index := maxint - 1
         End; {if}
         index:= succ(index);
      End; {while}
   End; {StoreVal}
{RtrvVal returns the value of location index in InterpTab}
Function RtrvVal(index : Integer) : Integer;

   Begin
      RtrvVal := InterpTab[index].value;
   End; {RtrvVal}

{DisplayStack displays the stack contents}
Procedure DisplayStack;
   Var
      linenum,
      inx : Integer;

   Begin
      linenum := 23;
      For inx := 1 to StackIndex + 1 Do
      Begin
         GotoXY (70,linenum);
         write ('        ');
         linenum := pred(linenum)
      End; {for}
      linenum := 23;
      For inx := 1 to StackIndex + 1 Do
      Begin
         GotoXY (70,linenum);
         write (Stack[inx]);
         linenum := pred(linenum)
      End; {for}
      readln
   End; {DisplayStack}
```

```
{push places a value onto the top of the run stack}
Procedure push (StackValue : Boolean);

   Begin
      If StackIndex < MaxStackSize Then
      Begin
         StackIndex  := succ(StackIndex);
         If StackValue Then
            Stack[StackIndex] := aTrue
         Else
            Stack[StackIndex] := aFalse;
         {endif}
         If showstack Then
            DisplayStack
         {endif}
      End {if}
      Else
         error := True
      {endif}
   End; {push}

{pushint places the value "StackInt" onto the top of the    }
{run stack                                                  }
Procedure pushint (StackInt : Integer);

   Begin
      If StackIndex < MaxStackSize Then
      Begin
         StackIndex := succ(StackIndex);
         Stack[StackIndex] := StackInt;
         If showstack Then
            DisplayStack
         {endif}
      End {if}
      Else
         error := True
      {endif}
   End; {pushint}
```

```
{pop returns the top value of the run stack}
Function pop : Integer;

   Begin
      pop := -1;
      If StackIndex > 0 Then
      Begin
         pop := Stack[StackIndex];
         Stack[StackIndex] := maxint;
         StackIndex := pred(StackIndex);
         If showstack Then
            DisplayStack
         {endif}
      End {if}
      Else
         error := True
      {endif}
   End; {pop}

{left returns a value for the first operand of the pcode   }
{entry                                                      }
Function left (nd1 : Integer) : Integer;

   Begin
      If nd1 = 0 Then
         left := pop
      Else
         left := RtrvVal(nd1);
      {endif}
   End; {left}

{right returns a value for the second operand of the pcode }
{entry                                                      }
Function right (nd2 : Integer) : Integer;

   Begin
      If nd2 = 0 Then
         right := pop
      Else
         right := RtrvVal(nd2);
      {endif}
   End; {right}
```

***** Proposition Interpreter *****

```
Procedure Propintr(    code  : codetype;
                   Var error : Boolean);
                     {calls errormsg, symval }
   Label
      Quit;

  Procedure StoreTestCase;

     Begin
        Readln (TextFile,name,value);
        While (not eof(TextFile)) and (name <> '&') Do
        Begin
           StoreVal (name, value);
           Readln (TextFile, name, value);
        End; {while}
        If name <> '&' Then
           error := True
        {endif}
     End; {StoreTestCase}
```

```
Procedure ProcessTestCase;

   Var
      index : Integer;

   Begin
      index := 1;
      While (code[index].op <> oend) and
            (index< CodeMax) Do
      Begin
         With code[index] Do
            Case op of
               oeq     : push(right(nd2) =  left(nd1));
               oneq    : push(right(nd2) <> left(nd1));
               olss    : push(right(nd2) >  left(nd1));
               ogeq    : push(right(nd2) <= left(nd1));
               ogrt    : push(right(nd2) <  left(nd1));
               oels    : push(right(nd2) >= left(nd1));
               oand    : push((right(nd2) = aTrue) and
                               (left(nd1)  = aTrue));
               oor     : push((right(nd2) = aTrue) or
                               (left(nd1)  = aTrue));
               oimp    : If (right(nd2) = aFalse) or
                            (left(nd1)  = aTrue) Then
                            push(True)
                         Else
                            push(False);
                         {endif}
               oeqv    : If right(nd2) = left(nd1) Then
                            push (True)
                         Else
                            push (False);
                         {endif}
               opush   : pushint(left(nd1));
               opushi  : pushint(nd1);
               olang   : If not nd1 = 1 Then
                            error:=True;
                         {endif}
            End; {Case}
         {endwith}
         index := succ(index);
      End; {While}
   End; {ProcessTestCase}
```

```
Begin {Procedure PropIntr}
   errormsg
      ('Do you want to see the stack as it changes? y/n');
   readln (answer);
   If answer = 'y' Then
      showstack := true
   Else
      showstack := false;
   {endif}
   Assign (TextFile, probname+'.ans');
   StackIndex := 0;
   InitStack;
{$I-}
   reset (TextFile);
   If IOResult > 0 Then
   Begin
      error := True;
      Goto Quit
   End; {if}
{$I+}
   Repeat
      StoreTestCase;
      If not (error) Then
      Begin
         ProcessTestCase;
         If pop <> value Then
            error := True;
         {endif}
      End; {if}
   Until eof(TextFile) or error;
Quit :
   Close(TextFile);
   If error Then
      errormsg
       ('***  your program is incorrect  ***')
   Else
      errormsg
       (' *** your program is correct ***')
   {endif}
End; {PropIntr}
```

***** Proposition Proof Interpreter *****

```
{general interpreter symbol table definitions }

   Var
      labels : CodeType;

Procedure proofint(    code  : codetype;
                   Var error : Boolean);
                      {calls errormsg, symval }

   Label
      Quit,
      EndReasons,
      EndTestCases;

{StoreTestCase stores a test case value and gets the next  }
{one                                                        }
Procedure StoreTestCase;

   Label
      Quit;

   Var
      index : Integer;

   Begin
      For index := 1 to 3 Do
      Begin
         Readln (TextFile,name,value);
         If not eof(TextFile) Then
            StoreVal(name,value)
         {endif}
         Else
         Begin
            error := True;
            Goto Quit
         End; {else}
      End; {For}
   Quit :
   End; {StoreTestCase}
```

```
{ EvalReason evaluates a reason and returns true/false }
Function EvalReason (code : CodeType) : Integer;

   Begin
      While (code[index].op <> oend) and
            (code[index].op <> owhy) Do
      Begin
         With code[index] Do
            Case op of
               oeq      : push(right(nd2) =  left(nd1));
               oneq     : push(right(nd2) <> left(nd1));
               olss     : push(right(nd2) >  left(nd1));
               ogeq     : push(right(nd2) <= left(nd1));
               ogrt     : push(right(nd2) <  left(nd1));
               oels     : push(right(nd2) >= left(nd1));
               oand     : push ((right(nd2)=aTrue) and
                                (left(nd1)=aTrue));
               oor      : push ((right(nd2)=aTrue) or
                                (left(nd1)=aTrue));
               oimp     : If (right(nd2)=aFalse) or
                             (left(nd1)=aTrue) Then
                             push(True)
                          Else
                             push(False);
                          {endif}
               oeqv     : If right(nd2) = left(nd1) Then
                             push (True)
                          Else
                             push (False);
                          {endif}
               opush    : pushint(left(nd1));
               opushi   : pushint(nd1);
               oarg     : ;
               ofun     : pushint(left(nd1));
               olang    : If not nd1 = 2 Then
                             error:=True;
                          {endif}
            End; {Case}
         {endwith}
         index := succ(index);
      End; {While}
      EvalReason := pop
   End;  {EvalReason}
```

```
Procedure StoreAxPremFact;
   Begin
      While (code[index].op <> oaxiom) and
            (code[index].op <> oprem)  and
            (code[index].op <> ofact)  and
            (code[index].op <> owhy) and
            (jnx <= codemax) Do
      Begin
         labels[jnx].op := code[index].op;
         labels[jnx].nd1 := code[index].nd1;
         labels[jnx].nd2 := code[index].nd2;
         jnx := succ(jnx);
         index := succ(index)
      End; {while}
      If (code[index].op = owhy) and (jnx <= codemax) Then
      Begin
         labels[jnx].op := oend;
         labels[jnx].nd1 := 0;
         labels[jnx].nd2 := 0;
         jnx := succ(jnx)
      End; {if}
   End; {StoreAxPremFact}

Begin {proofint}
   index := 1;
   If (code[index].op <> olang) and
      (code[index].nd1 <> 3) Then
   Begin
      error := True;
      Goto Quit
   End {if}
   Else
      index := 2;
   {endif}
{ store the axioms and premises }
   jnx := 1;
   While code[index].op <> ofact Do
      StoreAxPremFact;
   {endwhile}
   savefact := index;
{ increment past the fact to get to the reasons }
   While code[index].op <> owhy Do
      index := succ(index);
   {endwhile}
   index := succ(index);
   savereasons := index;
   Assign (TextFile, probname+'.ans');
{$I-}
   reset (TextFile);
   If IOResult > 0 Then
   Begin
      error := True;
      Goto Quit
   End; {if}
{$I+}
   Readln (TextFile,name,value);
   TrueCount := 0;
   While not eof(TextFile) Do
   Begin
      StoreTestCase;
      While code[index].op <> oend Do
      Begin
         whichnum := code[index].ndl;
         inx := 1;
{ find the axiom, premise, or fact }
         While labels[inx].ndl <> whichnum Do
            inx := succ(inx);
         {endwhile}
{ increment past the header record }
         inx := succ(inx);
         work := EvalReason(labels);
         If work = aFalse Then
            Goto EndReasons
         Else
            index := succ(index);
         {endif}
      End; {while}
      index := savefact;
      work := EvalReason {actually evalFact}(code);
      If work = aFalse Then
      Begin
         error := True;
         Goto EndTestCases
      End {if}
      Else
         TrueCount := succ(TrueCount);
      {endif}
EndReasons :
      index := savereasons;
   End; {while}
```

```
EndTestCases :
   If TrueCount = 0 Then
      error := True;
   {endif}
   If error Then
      errormsg
      (' *** your fact was not accepted ***')
   Else
   Begin
      errormsg
      (' *** your fact is accepted *** ');
      index := savefact;
      StoreAxPremFact {store the new fact }
   End; {if}
Quit :
End; {proofint}
```

***** Predicate Interpreter *****

```
Procedure Predintr(    code  : codetype;
                   Var error : Boolean);
                     {calls errormsg, symval }
   Label
      Quit, quit2;

Procedure StoreTestCase;

   Var
      inx : Integer;

   Begin
      inx := 1;
      Readln (TextFile,name,value);
      While (not eof(TextFile)) and (name <> '&') Do
      Begin
         If name = 'b' Then
         Begin
            B[inx] := value;
            inx := succ(inx)
         End {if}
         Else
            StoreVal (name,value);
         {endif}
         Readln (TextFile,name,value);
      End; {while}
      If name <> '&' Then
         error := True;
      {endif}
   End; {StoreTestCase}
```

```
Procedure ProcessCode;

   Begin
      With code[index] Do
         Case op of
            oeq      : push(right(nd2) =  left(nd1));
            oneq     : push(right(nd2) <> left(nd1));
            olss     : push(right(nd2) >  left(nd1));
            ogeq     : push(right(nd2) <= left(nd1));
            ogrt     : push(right(nd2) <  left(nd1));
            oels     : push(right(nd2) >= left(nd1));
            oand     : push((right(nd2) = aTrue) and
                             (left(nd1)  = aTrue));
            oor      : push ((right(nd2) = aTrue) or
                              (left(nd1)  = aTrue));
            oimp     : If (right(nd2) = aFalse) or
                           (left(nd1)  = aTrue) Then
                           push(True)
                        Else
                           push(False);
                        {endif}
            oeqv     : If right(nd2) = left(nd1) Then
                           push (True)
                        Else
                           push (False);
                        {endif}
            opush    : pushint(left(nd1));
            opushi   : pushint(nd1);
            olang    : if not nd1 = 3 then
                           error:=True;
                        {endif}
            orange   : Begin {InterpTab should have const }
                              {values in the field "value" }
                              {for jinx & kinx which was set }
                              {by calling addinx}
                           If nd2 <> 0 Then
                              highInx := nd2
                           Else
                              error := True;
                           {endif}
                           If nd1 <> 0 Then
                              lowInx := nd1
                           Else
                              error := True
                           {endif}
                        End;
            oindex   : pushint(B[boundVar]);
         End; {Case}
      {endwith}
   End; {ProcessCode}
```

```
Begin {Procedure PredIntr}
   errormsg
      (' Do you want to see the stack adjustments? y/n');
   readln (answer);
   If answer = 'y' Then
      showstack := True
   Else
      showstack := False;
   {endif}
   Assign (TextFile,probname+'.ans');
{$I-}
   reset (TextFile);
   If IOResult > 0 Then
   Begin
      error := True;
      Goto Quit
   End; {if}
{$I+}
   If code[1].ndl <> 3 Then
   Begin
      error := True;
      Goto Quit
   End; {if}
   If (code[2].op in [oall,oexist]) Then
   Begin
      theCase := code[2].op;
      boundVar := code[2].ndl
   End {if}
   Else
   Begin
      error := True;
      Goto Quit
   End; {else}
   index := 3;
   While code[index].op <> orange Do
   Begin
     ProcessCode;
     index := succ(index)
   End; {while}
   ProcessCode; {process the orange}
   If error Then Goto Quit;
   index := succ(index);
   StartOfCode := index;
```

```
{ process each test case }
   Repeat
      trueCount := 0;
      StoreTestCase;
      If error Then Goto Quit;
      low := RtrvVal(lowInx);
      high := RtrvVal(highInx);
      For boundVar := low to high Do
      Begin
         index := StartOfCode;
         While code[index].op <> oend Do
         Begin
            ProcessCode;
            index := succ(index)
         End; {while};
         newVal := pop;
         If newVal = aTrue Then
            trueCount := succ(trueCount);
         {endif}
      End; {for}
      If error Then Goto Quit;
      Case theCase of
         oexist : If trueCount = 0 Then
                        error := True;
                     {endif}
         oall   : If trueCount <> highInx-lowInx+1 Then
                        error := True;
                     {endif}
      End; {case}
      If (error and (value = aFalse)) or
         ((not error) and (value = aTrue)) Then
         error := False
      Else
         error := True
      {endif}
   Until eof(TextFile) or error;
Quit :
   If error Then
      errormsg
       ('*** your program is not correct *** ')
   Else
      errormsg
       ('*** your program is Correct *** ');
   {endif}
End; {Predintr}
```

THE DESIGN OF THREE INTERPRETERS:
PROPOSITION, PROPOSITION PROOF, & PREDICATE

by

Sarah R. Lauxman

B.S., Kansas State University, 1985

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

This report presents the design and implementation of a family of three related stack-machine interpreters, similar to the p-machine interpreted Pascal systems. The input for each interpreter is a series of pcodes. The pcodes represent single boolean expressions or series of boolean expressions. The interpreters execute the pcodes using test cases and evaluate the resulting boolean.

The three interpreters are proposition, proposition proof, and predicate; and correspond to three problem domains: proposition, proposition proof, and predicate. The boolean expressions are programs which are representations of story problems which belong to one of the three problem domains. The propositions are boolean (logical) expressions. The proposition proof problems are short proofs. With a few restrictions, the predicates are a subset of first order logic.