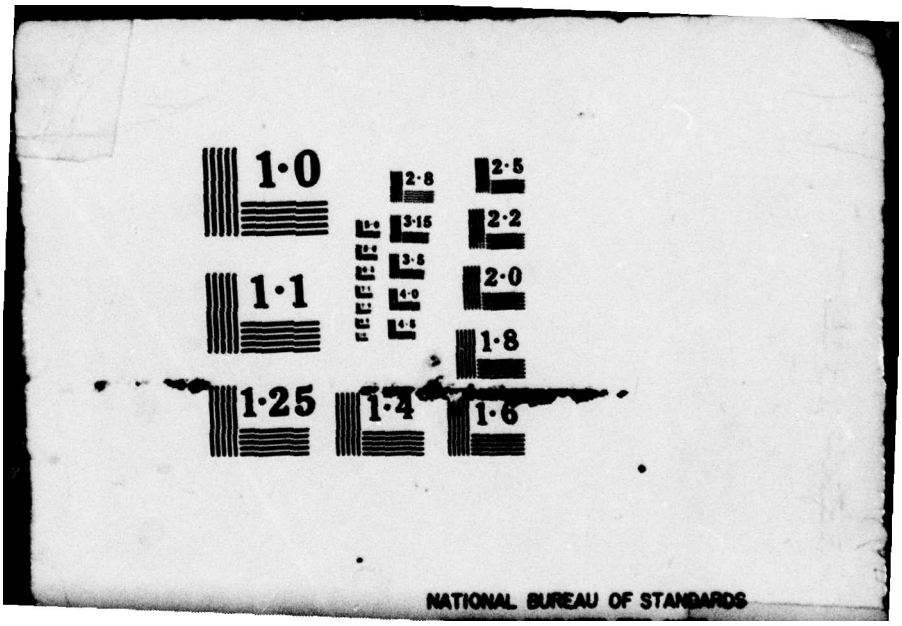
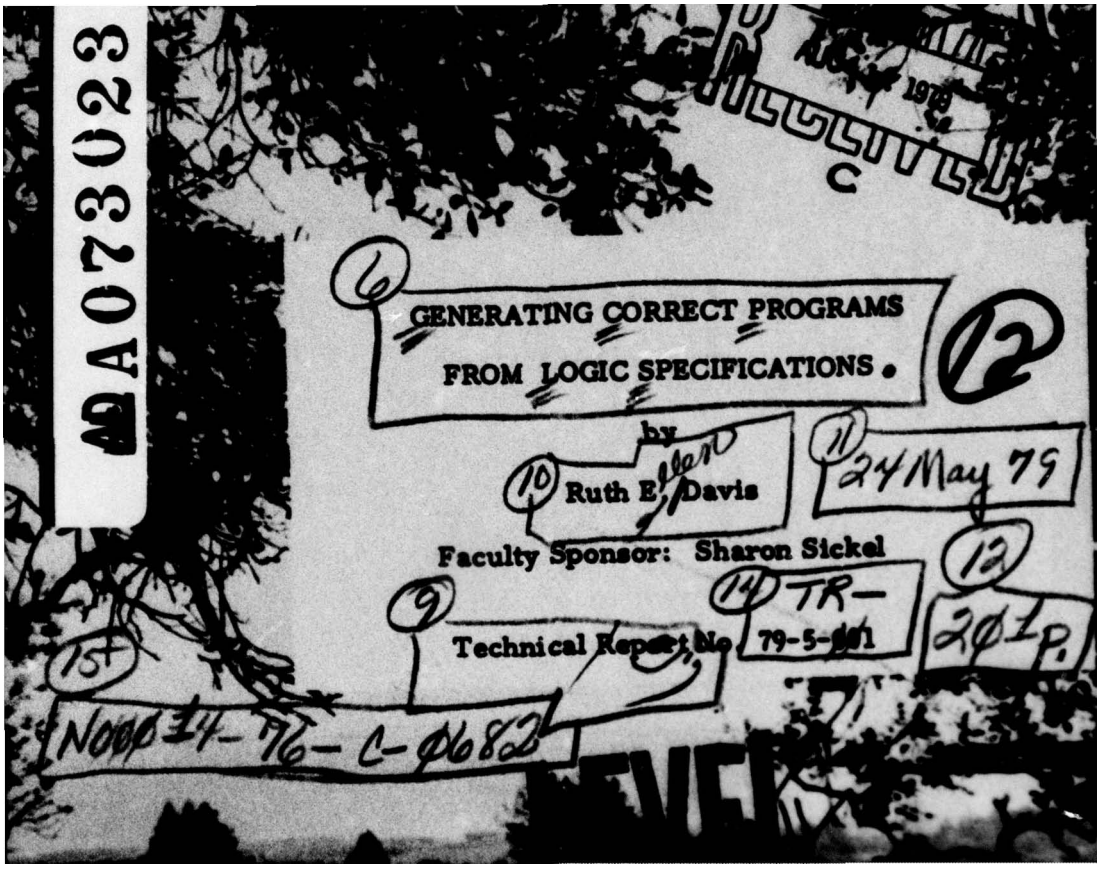
AD-A073 023 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
GENERATING CORRECT PROGRAMS FROM LOGIC SPECIFICATIONS.(U)
MAY 79 R E DAVIS N00014-76-C-0682
UNCLASSIFIED TR-79-5-001 NL


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
NATIONAL BUREAU OF STANDARDS


AD A073023
GENERATING CORRECT PROGRAMS
FROM LOGIC SPECIFICATIONS.
by
Ruth E. Davis
24 May 79
Faculty Sponsor: Sharon Sickel
Technical Report No. 79-5-001
201 P.
N00014-76-C-0682

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
Generating Correct Programs from Logic Specifications.	Technical
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Ruth E. Davis	N00014-76-C-0682
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Information Sciences University of California Santa Cruz, California 95064	
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Office of Naval Research Arlington, Virginia 22217	May 24, 1979
	13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report)
Office of Naval Research University of California 553 Evans Hall Berkeley, California 94720	Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program synthesis
automatic programming
program specification
logic programming

79 08 17 033

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We have designed and implemented a system that accepts logic specifications, generates algorithms in an intermediate language, and then translates these algorithms into programs in specific target languages.

The specification and intermediate languages are described in detail via their context free grammars and axiomatic semantics. It has been proved that the mappings preserve the axiomatic semantics of the programs.

We discuss the system as implemented, the requirements for extending it to new target languages, and further work suggested by the project.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

Generating Correct Programs From Logic Specifications

A Dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

INFORMATION SCIENCES

by

Ruth Ellen Davis

June 1979

The dissertation of Ruth Ellen
Davis is approved:

Sharon Sickel

Frank DeRemer

Carl Morgenstern

Dean of the Graduate Division

Copyright by
Ruth E. Davis
1979

Accession For
NTIS GRA&I
DDC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or special
A

Acknowledgements

I would like to acknowledge the contributions to this work of my thesis advisor, Professor Sharon Sickel; for her many ideas and criticisms, and her continual encouragement and support. I am grateful to the members of my committee, Professors Frank Deremer and Carl Morgenstern for their scrutinization and approval of various drafts. I would also like to thank John McCarthy, Lester Earnest and the Stanford Artificial Intelligence Laboratory for allowing me access to their facilities for the implementation of my research.

I am indebted to all of my friends and family, who never doubted that I would succeed, thus making it impossible for me to quit.

Finally, I would like to express my deep gratitude to the one person who really kept me going, my friend, colleague, and husband, John Allen, for his encouragement, patience, answers, and insights.

---

* The work reported herein was supported in part by the Office of Naval Research under contract grant number ONRN0001478C0682.

TABLE OF CONTENTS

1	Introduction	1
2	History	9
3	Specification Language	18
	3.1 Syntax of the Specification Language	22
	3.2 Semantics of the Specification Language	24
4	Logic Programming versus Synthesis	28
5	Intermediate Language	33
	5.1 Syntax of Intermediate Language	35
	5.2 Semantics of the Intermediate Language	36
6	Mapping From Input Specification to Intermediate Form	40
7	Correctness of the Mapping to Intermediate Form	48

8 Mapping the Intermediate Language to LISP 56

9 Correctness of the Mapping to LISP Program 59

10 Adding a New Target Language 70

10.1 Implementation Strategy for LISP 74
10.2 Implementation of Pascal 76

11 Implementation Notes 79

12 Conclusions and Further Research 83

BIBLIOGRAPHY 89

APPENDICES

14 Appendix A: Sample Specifications 94

15 Appendix B: Specification of a Program Synthesis System 113

16 Appendix C: Listing of the System 126

16.1 Listing of LISP Implementation 156
16.2 Listing of Pascal Implementation 172

Introduction

"Software is 5% of the USAF's budget, 6% of NASA's budget and is a 10 billion dollar a year industry (over 1% of the GNP)."

Standish [S 74]

"Of the estimated DOD software cost of $2.5 billion previously identified, 38% was for analysis, 15% for coding, and 47% for validation according to the Air Force study. ARPA is spending annually $3.5 million developing techniques to enable the computer itself to write and debug programs, given only a specificaion of the problem and the results desired."

Lukasik [L 79]

"The one invariant in the computer field - whether mainframe, mini, or micro - is increasing software costs. ...75 percent of the mainframe system dollar currently goes for software. ...Systems software typically represents 50 percent of the mini manufacturer's development budget (hardware costing 40 percent and services making up the remaining 10 percent). ...software costs tend to dominate in under-100-unit quantities - even on micros."

Davis [D 78b]

Software is expensive and too often bug ridden. "Debugging" is still the most commonly used method of increasing confidence in the correctness of a program. As indicated by the statistics above, validation of the correctness of programs is a difficult and expensive task. Many approaches have been proposed to ensure reliable programs, some dealing with the programming task and others with proving a completed program does what is intended. Verification techniques have been developed to prove that a program meets its specifications.

Several attempts have been made to provide automatic verification systems. Some of these systems are good at proving verification conditions for a specific problem domain, employing strategies that are geared to that domain; however, the ability to handle problems in a new subject domain typically requires extensive modifications of the system.

A much harder problem is discovery of the verification conditions to be proved. One must find assertions (statements of first order logic) that characterize the desired behavior of the program. It is difficult enough to read someone else's program and determine what is actually going on, let alone figure out what the programmer had intended to do. Even when verification is done by the original programmer, we are asking a lot. This person must be able (assuming a structured approach to the task) to move from logical assumptions about the problem domain (whether or not these are stated formally) to an implementation in a programming language, and then back again to the logical base as a source of assertions from which the verification conditions can be derived.

It is possible to generate some of these assertions automatically from the text of the program. Usually the programmer supplies the critical assertions on which his program was based, letting the machine fill in the necessary details. However, one might question the validity of this approach. A "proof of correctness" is really just a proof of equivalence of two specifications of the problem, one expressed in logic, the other in a programming language. If we derive the assertions (manually or automatically) from the program itself, then we lose the redundancy of the two specifications; i.e., we lose the basis of our confidence that the verification provides an indication of true correctness.

Furthermore, if verification fails, several explanations are possible. Perhaps the program is incorrect; perhaps the specifications are incorrect or incomplete; perhaps the verifier is incorrect or not sufficiently powerful to find a proof even though one does exist. If verification is successful, then again several explanations are possible. Perhaps we have a correct program; perhaps the verifier has made a mistake. Assuming that the verifier contains no bugs, we have a proof that the program meets its specifications, but the specifications may not be an accurate embodiment of what we had in mind.

Therefore several researchers, the author included, feel that a specification of the task should be written before a program is written to acomplish the task. The specification should be written in a high level language that enables one to describe what is to be accomplished, indicating functional relationships without having to consider computational details. Such a specification, in which only the abstract

description of the problem is required, is less prone to error than a typical programming language specification in which every detail of the computation must be provided. The problems of incomplete or incorrect specifications do not go away, however we claim they are more tractable with this approach.

Program synthesis is the generation of a computational specification of a problem or task from a descriptive specification. Again, we do not have the redundancy of two specifications, one descriptive and one computational, both written by the programmer, but we feel, as discussed above, that the descriptive specification is less prone to error. Several approaches to program synthesis are currently being investigated. We shall review them briefly here; more detail is given in the section on historical background.

The field of programming methodology has offered a great deal of assistance, pointing out ways to write a program. Most notable and generally accepted is Dijkstra's approach of "structured programming", also called "stepwise refinement" (Wirth), of the problem. Although intended as a discipline for humans to follow, it also provides guidelines that can be elaborated to direct programming by the computer itself, perhaps with human intervention.

Several investigators have taken a deductive approach to the stepwise refinement process. The computer is given specific rules by which it can rewrite statements syntactically while maintaining equivalent meaning. This kind of program synthesis is closely related to program transformation. The intent of a program transformation system is to make the given program more efficient while preserving its meaning. A

typical transformation is one that replaces recursion by iteration, although this does not always result in greater efficiency. In a synthesis system, a descriptive statement in a specification language is transformed into an operational statement in a programming language. The specification language may be a superset of the target programming language. The system successively refines a high level description into a lower level (computational) description.

Less formal techniques have also found favor. Natural language dialogues have been used to describe a problem to the computer. The user of this kind of system specifies the task at a general level, filling in the details as requested by the computer.

Still other approaches employ specification by example. Some systems synthesize programs from sample input-output pairs. For example, the pair

(a (b c) d) => (d (b c) a)

might indicate a procedure to reverse the elements in a list. It might also represent a procedure to switch the first and last elements of a list; thus, one must be sure to provide a sufficient set of examples to determine the desired function.

Another attempt at synthesis by example provides sample execution traces. For example, the sequence

(4 14) => (2 4) => (0 2) => 2

might suggest the Euclidean algorithm for finding the greatest common divisor of two integers.

Each of the techniques mentioned above have been investigated as a means to synthesize programs in a specific target language. It is our thesis that the synthesis of

an algorithm is a target-language-independent process. We have implemented a system to generate programs from logic specifications. The system is "reasonably" target-language-independent, as will be explained later on. The specifications are first translated into an intermediate language and then a program is generated from the intermediate form. We drop the word "synthesis" and use program "generation" instead to avoid any misunderstanding of the claims being made. The specification language for this system includes a subset of first-order Predicate Calculus known as Horn Clauses. The specifications we require are "descriptive" in that they specify the logic of the program without specifying the control, but they are also, in part, "computational" in that the Horn clauses could be "run" as programs given a complete theorem prover, or logic interpreter.

The system was implemented in MACLISP on a DEC KL10 at the Stanford Artificial Intelligence Laboratory. In this dissertation we describe the implementation and prove that it provides a valid way to derive correct programs. This is accomplished by proving that the top level mappings from specification to intermediate language to target program (the proof is for the mapping to LISP) are correct; the entire implementation is not proved correct.

We begin by giving a brief history of the approaches taken by other investigators in this area, and then describe the system invented by the author. We formally describe the specification language via a context-free description of the syntax (see page 22) and an axiomatic specification of the semantics (see page 24). Motivation for requiring particular items in the specification is provided as well.

The relationship between a specification and a program written for a logic interpreter is explored in Chapter 4,page 28).

We then describe the intermediate language in detail, again supplying a context-free grammar for the syntax (see page 35), and the axiomatic semantics (see page 36).

The mapping from specification language to intermediate language is described by means of a function *I*, for *internalize*, and we prove that this mapping preserves the axiomatic semantics of the specifications.

We then describe the mapping from the intermediate language to LISP and prove that the semantics is again preserved by the translation.

In Chapter 10, we describe how to extend the system to handle generation of programs in more languages. The additions required for each new language are referred to as a "back end" for that language. The implementation of the "back end" for LISP and that for Pascal are also discussed.

We include some notes on the implementation and then discuss the conclusions that can be drawn from this effort, as well as topics of further research that were suggested by the project.

The appendices include several sample specifications and the programs generated from them, the specification of a program generation system, and the listing of the entire system along with the code that implements the "back end" for LISP and that for Pascal.

This system described is unique in its ability to generate programs in more than

one language. The recipe provided explains how to construct the additions required for any new language. Given the back-end describing a particular target language, we can synthesize the system itself in the language, making it immediately portable. More importantly, it provides a means of obtaining correct programs that is decidedly less painful than verification. The user is still required to specify the logic of a program, but the language used in the specification frees the user from concerns of representation and implementation.

The sobering fact remains that regardless of how (or when) we arrive at specifications for a program, we can have no guarantee of their correctness (in the sense that we have both a true and complete specification of the problem we had in mind), and any attempt at verification of a program is simply a proof of equivalence of program and specifications. We simply do the best we can to increase our confidence that our programs accomplish their intended purpose.

History

Computer scientists have always looked for ways to make the programming task easier and less prone to error. The natural way to accomplish this is by using the computer itself as much as possible to do its own coding. The theoretical foundation for automatic programming was established by Kleene [K 52] in the 1940's. Kleene proved that if the existence of a number satisfying certain properties can be proven in a formal intuitionist system, [1] then the definition of a function computing that number can be extracted from the proof. The first major attempt at automatic programming was the development of FORTRAN.

> "If it were possible for the 704 to code problems for itself and produce as good programs as human coders (but without the errors), it was clear that large benefits could be achieved. ...The goal of the FORTRAN

[1] An intuitionist system allows only constructive proofs. See [K 52].

> project was to enable the programmer to specify a numerical procedure using a concise language like that of mathematics and obtain automatically from this specification an efficient 704 program to carry out the procedure."
>
> Backus, et. al. [B 57]

In the 1960's several high level languages were introduced as means of specifying problems to the computer in a way more natural to us, letting the machine do the coding. Over the years, our concept of "automatic programming" has changed. We no longer consider a compiler an automatic program synthesizer.

J. R. Slagle [S 65] applied his question answering program "DEDUCOM" to the task of generating programs. The relation between input and output was expressed in predicate calculus. His technique was to prove a theorem and write a program by keeping track of the substitutions made for certain crucial variables in the course of the proof.

A similar approach was described by Waldinger [W 69]. He extended the technique allowing branches and loops to be written. Again, specifications for the program were described in Predicate Calculus as a relation between input and output variables. Mechanical theorem proving techniques were used to generate a constructive proof of the existence of output values satisfying the specifications. A program was then extracted from the proof.

Since 1970, several approaches to program synthesis have been investigated. The resolution theorem proving approach proved to be impractical, requiring too much space while considering all possibilities. Lee and Chang [LC 74] proposed to

overcome the memory saturation problem with an interactive system based on the concept of structured programming. Using the technique of stepwise refinement, a program was generated in terms of subprograms until each subprogram became an "atomic program", i.e. executable. At each step, some appropriate information would be selected by the user for the computer to use to generate a subprogram. In this way, the computer only had to handle a small amount of information.

Many researchers abandoned resolution methods entirely in favor of deductive systems with many rules of inference. Buchanan developed a system [B 74] based on the program verification formalism of Hoare [ILL 75]. Input to the system was given in "frames" composed of assertions, state descriptions, axioms and rules. A rule could be: a primitive procedure with preconditions and postconditions specified; a definition, stating the equivalence of two assertions; or an iterative rule specifying conditions that, if satisfied, would justify the assembly of a "while" loop to achieve the associated goal. The input specifications were rather complicated. An iterative rule involved specifying a name, a basis assertion, a loop invariant, an iteration step assertion, an iterative goal, a loop control test, and an output assertion (the last two could possibly be the same as the iterative goal). It was also possible to give advice to the system interactively, to guide the synthesis process. This system could automatically generate conditional statements as well.

Dershowitz and Manna [DM 75] discussed formalization of several programming techniques involved in structured programming, and demonstrated the use of these rules by hand-synthesizing programs. Manna and Waldinger [MW 77c]

have implemented such techniques in "DEDALUS", an experimental program synthesis system. They have attempted to make life easier for the user by simplifying the input required. They have not, as yet, attempted to describe completely their specification language; it is a superset of the target language containing quantifiers and several high-level constructs from the subject domain. Hundreds of transformation rules are available, embodying a great deal of knowledge about the domain for which programs are to be synthesized. The transformations include rules for recursion formation, conditional formation, and procedure formation. A goal given as input is transformed into subgoals until a primitive program to accomplish it is derived. Strategic controls are used to choose among possible synthesis paths.

Burstall and Darlington [BD 75] described a formal system for manipulation and optimization of recursive functions. The language they use is that of recursion equations. Darlington [D 75] extended this language to include set notation (having been influenced by the work of Manna and Waldinger), and described the application of the transformation system to the problem of program synthesis. (The difference between program transformation and program synthesis lies in the degree of abstraction in the specification one starts with.) Non-trivial algorithms have been derived manually using the transformation rules of the system, however cleverness is still needed at some points to determine which rules to apply.

Clark and Sickel [CS 77] describe the process of deriving computational logic programs (defined by Sickel [S 77a]) from axiomatic specifications. The process was investigated using hand synthesis of programs, but an interactive system was intended

to be implemented that would become more automatic as the synthesis methodology was refined. Clark and Darlington [CD 78] further this methodology using a compromise notation of logic and recursion equations, making the results applicable in both formalisms. They describe the synthesis of recursive function definitions from axiomatic specifications. Sickel [S 77a] also described a methodology for continuing the process down to an executable form in a "conventional" programming language. By doing a theorem proving style of analysis of the logic program one can derive a tree representing all possible proofs of the program taken as a theorem. A regular expression is used to describe a computation path representing all proofs of the theorem. This computation path may then be mapped into a program in the target programming language.

Observing that the structure of a program is often determined by the structure of the data it operates on, von Henke [H 75] organized knowledge about the data domain and represented it in such a way that it can directly assist a system in constructing programs. He used LCF (Logic for Computable Functions), extended to include terms for expressions involving sets and bounded quantification, as the problem specification language. The fact that every LCF term also has an interpretation as a computational rule for the function denoted by it allows the term to be regarded as a program. Data type definitions are used to generate "characterizing" functions (identity on the type defined and undefined elsewhere) which can then be abstracted into functionals for homomorphic and endomorphic extension. For instance, a predicate to recognize the type could be described as a homomorphic

extension into truth values. A function to accomplish substitution, "replace free occurrences of *v* in *e* by *t* after renaming bound variables in *e* so that no free variable in *t* becomes bound in the modified *e*", can be expressed as an endomorphism on the data type "expressions".

All of the approaches mentioned above deal with formal systems in which the proofs of equivalence of program and specification can be carried out. Many researchers have experimented with systems that automatically write or modify programs from partial specifications in an ambiguous language (subsets of English), making correctness more difficult to arrive at. Heidorn [H 76] reviews four such projects that use natural language dialogues with the machine for specification of the problem. Most such projects limit the area of application severely. In his own research at the NPS (Naval Postgraduate School, Monterey, Ca.) Heidorn used a restricted form of English as input and generated GPSS programs as output. The system used hundreds of decoding and encoding rules and was designed for generation of programs to do queueing simulations. These it did well with the author of the system as user.

Another project aiming at natural language specification is being carried on at the Information Sciences Institute of the University of Southern California. This system, called SAFE [BGW 77], is intended to be independent of the problem domain, and consists of three phases: "domain acquisition" [GBW 78], "planning" [WBG 77], and a phase to produce the final program. The input to the system is a (manually) parenthesized natural language program description retaining most semantic

ambiguities of natural language but avoiding its syntactic ambiguities. The phases deal with the data and operation structure, the program and control structure, and the program variable and parameter structure, respectively.

Lenat [L 75] studied the problems involved in synthesizing large LISP programs requiring several hours of user-system interaction time to generate natural language specifications for the problem. The problem domain was inductive inference programs. The system was made up of BEINGs, experts on various topics (such as coding, probability, or contradiction), capable of asking and answering questions of the user or other BEINGs. It was found that to be successful a user had to be "familiar with LISP, well-grounded in computer science, and have the input-output behavior clearly in mind." The system was constructed with particular dialogues in mind. Problems pointed out by the experiment were the inflexibility of the system to new dialogues, its dependence on user reliability (no errors allowed), and the system's inability to accept new high level domain-specific knowledge (these additions had to be made through modifications of the system itself).

Another approach is synthesis of programs by example. Hardy [H 75a] describes a system that generates LISP functions from a single input-output pair. He deals only in the domain of list-manipulating functions and claims that "despite the fact that there are infinitely many functional extensions of the input-output

('iopair'): (A B C D) = > = ((A) (B) (C) (D))

there is only one function that would be regarded as the 'obviously' intended one." This is certainly true, and he has quite a few examples that work as one might expect.

However one can imagine several functions incapable of being completely characterized by a single input-output pair, e.g., any function which operates differently as the result of a test on the data it recieves as an argument (if the first element of an integer list is even then ...).

Summers [S 77b] uses several sample input-output pairs to synthesize LISP functions. This is acomplished by a series of transformations from a set of examples to a program. Programs are synthesized in a subset of LISP using the primitive functions *car*, *cdr*, *cons*, and *atom*, and the control structures of recursion, functional composition, and the conditional expression. The current system, THESYS, is able to derive programs with at most a single recursive call. A technique called "differencing" is used to set up equations that can be rewritten as a set of recurrence relations which are then used to find the recursive program satisfying the examples. Another technique, "variable addition", is used to generalize on a set of examples with the hope of enabling differencing where it was not applicable before.

Rather than supply example input-output pairs, Ulrich and Moll [UM 77] advocate supplying an entire program as an example and establishing an analogy that the computer can use to derive another program. By extending an analogy, a given program may be used to generate another program solving a different but analogous problem. The analogy formation process works with the proof of the known program rather than with its code. The proof is attempted for the new domain and only altered when a step is not valid in this domain. A change in the correctness proof causes a change in the code. The initial analogy must always be specified by the user, making the solution of subproblems somewhat awkward.

Biermann and Krishnaswamy [BK 76] construct programs from sample computation traces. They have shown that if there exists a program capable of executing the given trace, then their system will find that program (or one equivalent to it). This approach assumes the user has an algorithm already in mind (to supply the computation trace).

Several of the above approaches are brought to bear in a project lead by Green [G 77] at Stanford University. The PSI program synthesis system consists of two phases: an acquisition phase and a synthesis phase. The specification is accomplished through a dialogue with the user, using English descriptions, examples, and traces. The acquisition phase is made up of a parser-interpreter, a trace and examples inference system, a dialogue moderator, a domain expert, and a model builder. The synthesis phase involves the interaction of a coder and an efficiency expert to refine the program model into an efficient executable program. The PSI system is being extended as the group attacks different problem domains.

Specification Language

The input to the system is a sequence of definitions. A target definition specifies the target language to be used, which may be changed during a session by simply supplying another target definition. The other kinds of definitions are function, type and generic.

A function definition has six parts, a name, an input pattern, a formal parameter list, a precondition, a postcondition, and a body.

For example:

```
function Fact
input pattern? (1 0)
parameter list? (x y)
precond? Integer(x, true) ∧ ≥(x, 0, true).
postcond? Integer(y, true) ∧ >(y, 0, true).
body? Fact(0, 1)
    Fact(z, w) ← Subl(z, z1), Fact(z1, w1), *(z, w1, w).
```

The verbosity in the indicators is endurable since the system types out the question asking for each part of the specification following the function name. The name part is self-explanatory. An input pattern is a list of 1's and 0's indicating which of the arguments of the function being defined are expected to hold input values and which are used to carry values produced during the computation of the function, respectively.

A formal parameter list is required to match up the appropriate arguments for the precondition and postcondition specifications. As we will see, the formal parameters are independent of the specification of the body.

The precondition is a well-formed-formula of predicate calculus, defined on the input variables, that is true only if the function is defined for those input variables. By specifying precisely the domain of the function we can guarantee termination of each program. The precondition is more than just a type declaration in the usual sense, such as *Integer* or *Real*, it may also provide information such as "$x \geq 2$" assuming that x is an input variable. Similarly, the postcondition is a predicate formula on the output variables that specifies the range of the output. Taken together, the precondition and postcondition specify the functionality, i.e. the domain and range, of the program being defined. The range given may in fact be larger than the actual range of the function, but the domain given must not include any extraneous elements.

A body definition is a set of Horn clauses that describes the function, usually recursively, in an axiomatic way. Each Horn clause is an implication, stating that the goal can be asserted if a set of subgoals can be satisfied. In writing a body definition we are asserting that each of these implications is true.

Formally, a Horn clause is a disjunction of literals (atomic formulas, such as "$P(x, 2, y)$", or negated atomic formulas) in which at most one literal is positive (not negated). The implication form is derived from the fact that $(A \vee \neg B \vee \neg C) \equiv (A \leftarrow B \wedge C)$. There are four kinds of Horn clauses:

1) one with non-empty antecedent and consequent;
2) one in which the antecedent is empty and the consequent is non-empty;
3) a clause with non-empty antecedent and an empty consequent; and
4) an entirely empty clause.

Only the first two of these types of clauses are used in function definitions. The first is an implication as mentioned above, the second is considered an assertion. (The assertion arises from the fact that we can consider the form of the implication to be a conjunction implying a disjunction, an empty conjunction is interpreted as true, thus asserting the implication "$true \rightarrow A$" is the same as asserting "A".) Kowalski [K 74] describes the interpretation of Horn clauses as a programming language. We will discuss the differences between Logic programs and the specifications required by this system in a later section; see page 28.

A type definition is essentially a definition of the predicate that recognizes occurrences of the type. In this sense it has the same form as any other function definition. The body of a type definition has the same form as that of a function definition. For the sake of consistency, we require that each type predicate have an output variable, just as any other function definition would. Although one may not usually think of a predicate as needing an output variable, since the evaluation of the predicate succeeds if and only if the answer is "true", the inclusion of an explicit output variable allows us to distinguish when it is possible to give an answer of "false" from the case in which we simply cannot determine that the answer is "true".

For example,

type Set
body? Set(Mt-set, true)
Set(Add-elem(y,X), true) ← Set(X, true), Member(y, X, false).

Much of the information is implicit in a type definition. The input pattern is always *(1 0)*, the formal parameter list is any list of two variables, the precondition is *TRUE*, and the postcondition is simply that the output variable ends up with a truth value.

A generic function is one in which the input pattern is allowed to vary. For example, one might wish to define a function *"Concat(x, y, z)"*, meaning "z is the result of appending x and y". If defined as a generic function we can use *Concat* in defining other functions whenever we have two of its arguments available and wish to compute the third.

The definition of a generic function includes its name, a formal parameter list (again for the purposes of the preconditions and postconditions), a list of choices specifying how the function may be used, and possibly more than one body definition. The general form of a generic specification is [2]:

"generic" name
"parameter list?" id-list
"choices?"
input-pattern$_1$
  "function name?" funname$_1$
  "precond?" precondition$_1$
  "postcond?" postcondition$_1$
  "body-name?" bodyname$_1$
...
input-pattern$_n$

[2]The subscripts used are for clarity in the example; they are not part of the syntax.

"function name?" $funname_n$
"precond?" $precondition_n$
"postcond?" $postcondition_n$
"body-name?" $bodyname_n$ "."

"body-defs:"
$name_1$ "?" $h\text{-}clause_1$ "."*
...
$name_k$ "?" $h\text{-}clause_k$ "."*

A choice is made up of an input pattern, a name to be associated with this particular style of call on the function, domain and range specifications, and a body name indicating the function body to be used. A body definition associates a body name with a definition (i.e. a set of Horn clauses).

3.1 Syntax of the Specification Language

A context-free description of the input language is as follows:

PHRASE STRUCTURE:

input ::= definition "."*

definition ::= fun-def | type-def | gen-def | target-def

fun-def ::= "function" name
"input pattern?" input-pattern
"parameter list?" id-list
"precond?" precondition
"postcond?" postcondition
"body?" h-clause "."*

type-def ::= "type" name
"body?" h-clause "."*

gen-def ::= "generic" name
"parameter list?" id-list
"choices?" [choice "choices?"] "."*
*"body-defs:" body-def**

target-def ::= "target" target-language

target-language ::= name

choice ::= *input-pattern*
"*function name?*" *name*
"*precond?*" *precondition*
"*postcond?*" *postcondition*
"*body-name?*" *name*

body-def ::= *name* "?" *h-clause** "."

input-pattern ::= "(" *zero-or-one** ")"

zero-or-one ::= "0" | "1"

name ::= *identifier*

id-list ::= "(" *name** ")"

precondition ::= *disjunction* "."

postcondition ::= *disjunction* "."

disjunction ::= *conjunction* | *conjunction* "∨" *disjunction*

conjunction ::= *literal* | *literal* "∧" *conjunction*

literal ::= "True" | "T" | *pred-app* | "(" *disjunction* ")"

pred-app ::= *name arglist*

arglist ::= "(" *arg** ")"

arg ::= *fun-app* | *variable* | *constant*

fun-app ::= *name arglist*

variable ::= *identifier*

h-clause ::= *goal* "←" *subgoals* | *goal*

goal ::= *pred-app*

subgoals ::= *pred-app* ["," *pred-app*]*

constant ::= *number* | *string* | "*true*" | "*t*" | "*false*" | "*f*"
| "*undef*" | "(" ")" | *quoted-const*

string ::= *dblquote* [*anychar* | *punctuation* | *dblquote dblquote* | " "]* *dblquote*

punctuation ::= "," | "." | "'" | "`" | ";" | ":" | "/" | "]" | "(" | ")"

variable ::= *identifier*

quoted-const ::= "'" *exp*

exp ::= *identifier* | "(" *elem** ")"

elem ::= *constant* | *variable* | *exp*

LEXICON:

number ::= *integer* | *real*

integer ::= *digit-seq* | *sign digit-seq*

mantissa ::= *integer* "." *digit** | *sign* "." *digit-seq* | "." *digit-seq*

real ::= *mantissa* | *mantissa* "e" *integer*

digit-seq ::= *digit digit**

digit ::= *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9*

sign ::= "+" | "-"

identifier ::= *anychar* | *anychar identifier*

where *anychar* ∈ {≡,∩,∪,⊂,⊃,⊼,+,-,*,⊄,//,≤,≥,<,>,≠,=,∝,ß,ε,λ,π,∀,∃,/,?,
q,w,e,r,t,y,u,i,o,p,a,s,d,f,g,h,j,k,l,z,x,c,v,b,n,m,Q,
W,E,R,T,Y,U,I,O,P,A,S,D,F,G,H,J,K,L,Z,X,C,V,B,N,M}

and *dblquote* ∈ { " }

3.2 Semantics of the Specification Language

The semantics of a function specification:

"function" name
"input pattern?" input-pattern
"parameter list?" id-list
"precond?" precondition
"postcond?" postcondition
*"body?" h-clause** "."

informally, is that for all possible instantiations of the formal parameter list, if each parameter designated as an input by the input pattern has a value bound to it and the value of the precondition evaluated on the inputs is true (i.e., the inputs lie within the domain of definition of the function), then the conjunction of each of the Horn clauses and the postcondition is true. Expressed as a formula in first order logic:

$\forall \bar{x}$ defined[inputs[S[*id-list*], S[*input-pattern*]]] ∧ S[*precondition*] →

$$S[h\text{-}clause^*] \wedge ((S[name]\ S[id\text{-}list]) \rightarrow S[postcondition])$$ [3]

where:

the semantic function S maps elements of the specification language onto their denotations in first-order logic. Due to the similarity of the languages involved, this mapping is the identity mapping for the *id-list*, *input-pattern*, and *name*. The denotations of the *precondition* and *postcondition* are the obvious formulas of first-order logic. Elements of the semantic domain will be given in bold type whenever a distinction is desired.

$\bar{x}$ represents all variables in S[*id-list*]

inputs[S[*id-list*], S[*input-pattern*]] is the list of input parameters, i.e., those parameters in S[*id-list*] corresponding to 1's in S[*input-pattern*]

defined[u] is true iff every element of the list u has a value that is completely defined (contains no free variables)

S[*h-clause**] is the conjunction of the semantics of the individual horn clauses. The semantics of each horn-clause is that for all variables mentioned, the conjunction of the right-hand-side implies the left-hand-side.

S[*name id-list*] is the denotation of the application of S[*name*] to S[*id-list*] in first-order logic.

For example:

```
function Fact
input pattern? (1 0)
parameter list? (x y)
precond? Integer(x, true) ∧ ≥(x, 0, true).
postcond? Integer(y, true) ∧ >(y, 0, true).
body? Fact(0, 1)
      Fact(z, w) ← Subl(z, z1), Fact(z1, w1), *(z, w1, w).
```

has the semantics of the associated first order logic expression:

```
∀x,y [ defined[(x)] ∧ Integer(x, true) ∧ ≥(x, 0, true)  →
     Fact(0, 1)
       ∧ ∀z,z1,w,w1 [ [Subl(z, z1) ∧ Fact(z1, w1) ∧ *(z, w1, w)  →  Fact(z, w)] ]
          ∧ [ Fact(x, y)  → Integer(y, true) ∧ >(y, 0, true) ] ]
```

The semantics of a type specification:

```
"type" name
"body?" h-clause* "."
```

---

[3] We assume that the logical operators have the following precedence relationships, from tightest to least binding: ¬, ∧ and ∨, →, ≡ or <->.

is given by the first order logic formula:

$$\forall x \ \text{defined}[\ (x)\ ] \ \rightarrow \ S[h\text{-}clause^*] \wedge [\ S[name\ (x, y)] \rightarrow \text{boolean}(y,\text{true})]$$

The antecedent of the implication guarantees that we have an input value, the consequent asserts the conjunction of the Horn clauses making up its body and the fact that the result of a type predicate is a truth value. The precondition is "True" (and therefore need not even appear in the antecedent of the implication) since we want to allow anything at all as input to a type predicate, the postcondition need not be stated explicitly in the specification since it is always the same.

The semantics of a generic function specification [4]:

"generic" name
"parameter list?" id-list
"choices?"
$input\text{-}pattern_1$
  "function name?" $funname_1$
  "precond?" $precondition_1$
  "postcond?" $postcondition_1$
  "body-name?" $bodyname_1$
...
$input\text{-}pattern_n$
  "function name?" $funname_n$
  "precond?" $precondition_n$
  "postcond?" $postcondition_n$
  "body-name?" $bodyname_n$ "."
"body-defs:"
$name_1$ "?" $h\text{-}clause^*_1$ "."
...
$name_k$ "?" $h\text{-}clause^*_k$ "."

is given by the conjunction:

$$\bigwedge_{i=1}^{n} \forall \bar{x} \ [\ (S[funname_i\ id\text{-}list] \rightarrow S[name\ id\text{-}list]) \wedge (\text{defined}[\ \text{inputs}[S[id\text{-}list],\ S[input\text{-}pattern_i]]\ ] \wedge S[precondition_i] \rightarrow S[\ \text{H-C}[bodyname_i]\ ] \wedge (S[funname_i\ id\text{-}list] \rightarrow S[postcondition_i])\ )\ ]$$

---

[4]Note again that the subscripts are not part of the syntax

where:

H-C[*bodyname*$_i$] is the *h-clause*$^*_i$ associated with *bodyname*$_i$,
the *input-pattern*$_i$'s are distinct,
the *funname*$_i$'s are distinct,
the *bodyname*$_i$'s need not be distinct,
the *name*$_i$'s are all distinct and are all the different *bodyname*$_i$'s listed above,
and S is the identity mapping on *bodyname*'s and *funname*'s.

Thus the generic specification has the effect of defining several diferent functions and associating them all with a "generic" name that can be used when one does not wish to bother with using a different name every time the function is called in a different way, that is, with a different input-pattern.

The semantics of a target definition is simply the ordinary Hoare rule for assignment:

P[name/target] { *target name* } P

Using this specification language we can describe an algorithm for unification of two lists of terms as follows:

```
function Unify
input pattern? (1 1 0)
parameter list? (t1 t2 s)
precondition? List(t1, true) ∧ List(t2, true).
postcondition? Substitution(s, true).
body?  Unify( (), (), ())
       Unify( (), cons(x,u), undef)
       Unify(cons(x,u), (), undef)
       Unify(cons(x,t1), cons(y,t2), s) ← Unifyterms(x, y, s1),
                                          Mk-subst(s1, t1, t2, newt1, newt2),
                                          Unify(newt1, newt2, s2),
                                          Compose-subs(s1, s2, s).
```

Logic Programming versus Synthesis

Predicate logic can be viewed as a powerful, high level, nondeterministic programming language. Why not simply program in logic instead of bothering with a system to obtain programs in some other language? For one thing, logic is often more powerful than we need, and this power is not free. The full backtracking abilities required by an implementation of logic can be costly in time and space. Also, some languages are better suited to particular problems than are others. Although the style of program generated by the system proposed here is inherited from the specification and therefore similar in all languages, a program transformation system may be constructed that could optimize programs in a specific language to take advantage of its special features. The main purpose of the synthesis system is to see that we end up with correct programs. In this section we look at logic as a programming language, and then as part of the specification language for this system.

The completeness of the Horn clause subset of predicate logic as a programming language has been proven by Andreka and Nemeti [AN 76]. The operational semantics has been found [EK 74] to be part of the proof theory of predicate calculus and thus closely related to the axiomatic semantics. The operational semantics involves consideration of all possible derivations from the axioms. This implies, among other things, the ability to compute relations, not just functions. For example, the operational semantics of the atomic formula *Times(x, y, z)*, would be described as follows:

$$(a, b, c) \in D_{op}[Times] \text{ iff } (a, b, c) \in \{ (x, y, z) \mid \vdash_{\mathcal{X}} Times(x, y, z)\}$$

where $\mathcal{X}$ is the theory of predicate calculus augmented by the definitions given as axioms. A computation of *Times(3, 4, x)* would find only one answer, with x=12. However, a computation of *Times(x, y, 12)* would involve several derivations: *Times(1, 12, 12)*, *Times(12, 1, 12)*, *Times(2, 6, 12)*, *Times(6, 2, 12)*, *Times(3, 4, 12)*, and *Times(4, 3, 12)*, (in some order). The computation of *Times(3, y, z)* would never terminate since there are an infinte number of derivations possible.

There is no distinction made between input and output. A given tuple is in a particular relation or not. This facet of logic programming is particularly useful in data base applications [E 78].

There is no order of evaluation implied by a logic program. Its operational semantics involve all derivations without indicating which should be attempted first. Thus, a logic program, together with a call on it, determines a computation tree but says nothing of which path to follow. Under these circumstances, to prove termination of a logic program, one must prove the tree is finite.

In any given implementation of a logic interpreter things are not that bad. The order of evaluation is deterministic, so the tree is always traversed in a predictable way. Even so, if all derivations are to be attempted, termination occurs only for finite trees. However, termination can be proved under some circumstances.. For example, if only a single answer rather than *all* answers is desired, then one may be able to prove termination for a particular order of evaluation with respect to specific argument positions being designated for inputs.

In the current system we have chosen to define functions rather than relations, in the sense that we want a single unique answer to a question rather than all possible answers. This decision is based on the belief that a programmer knows when *the* answer, *any* answer, or *all* answers to a problem is desired and can design the program to ask for such explicitly. Asking for the answer "the set of all framices that foo" is a request for a single unique answer.

The specification of a function includes a Horn clause description. The specification is translated into a deterministic algorithm expressed in an intermediate language. Currently, clauses are tried in the order given as are subgoals within a clause. An automatic ordering is under consideration by Mike Franusich at the University of California at Santa Cruz. Some analysis can be done to ensure, for example, that termination cases are attempted first, and subgoals that produce values are called before other subgoals that need those values as input.

We distinguish between input and output variables for two reasons. First, given which values are expected to be available and which are to be computed, we can often

prove termination when in the general case we may not be able to (for instance, in the *Times(x, y, z)* example discussed earlier). Also, perhaps more importantly, we believe that programs tend to be based on the construction of the variables one expects to compute. The programmer is more likely to see a task as deriving output information from given input information than as a non-directional exploration of the relationship between the two. Thus, even if the program terminates when inputs and outputs are interchanged, the computation may become horribly inefficient.

Programs can be written that are "nicely" invertible, meaning that one direction of computation is about as efficient as another. In logic programs invertibility is more general than simply swapping of input and output variables; we are dealing with n-tuples not ordered pairs. Thus, for an n-ary predicate there are 2^n different ways of designating input and output variables. Sickel [S 78] describes j-invertibility, refering to the jth variable of the n-tuple. She presents some guidelines for constructing invertible functions, and describes algorithms to test the invertibility of programs.

Since it can often be useful to describe one procedure that may be used in different ways, we have included the concept of a "generic function"[5]. We wish to

---

[5]According to the dictionary, "generic" means "applicable or referring to all the members of a genus or class". A "generic" function is mentioned elsewhere in the literature as a function whose specific form is determined by the data type of its arguments. We are extending this notion to include functions whose "input patterns" (determining which formal parameters are to be expected as input when a function is

allow the convenience of invertible functions while requiring the programmer to be aware of the possibilities in the program. Thus, we require that one list the different calling styles by input-pattern and indicate a function body (set of Horn clauses) to be used in each case. A predicate with six different (useful) calling styles may use the same logic program as the specification for three input-patterns, another for two input-patterns, and a third for the last. Thus we get the best of both worlds; we know enough to maintain the guarantee of termination, and the user gains the flexibility afforded by a generic function in the specification of other programs. The system attaches a different name to each function body and determines from context which is appropriate in a given instance of the generic.

---

called) may vary. The implementation of the generic chosen by type rather than input pattern is suggested as an extension of the current system .

Intermediate Language

The intermediate language form of function definitions is strictly for use by the system. The user never sees or programs anything in the internal form. The auxiliary specifications (all but body specifications) are essentially unchanged; their representation is slightly different but exactly the same information is expressed. Thus, the only really interesting part of the intermediate form is its treatment of the body specification.

The body of a function in internal form is a "backtracking-conditional" ("bktrkcond" for short). The name is slightly too general as only a restricted, well behaved, kind of backtracking is allowed. Each clause of the specification becomes an "alternative" in the bktrkcond. An alternative consists of a "match"-part, which is an argument list to be matched against the actual parameters of a function call, and a

"try" part consisting of the subgoals to be accomplished. If the match-part of an alternative is accomplished, then the actual call on the function is an instantiation of the head (or goal) part of the clause from which the alternative was derived. Thus, according to the specifications, we can assert this call if it is possible to accomplish all the subgoals. If the match-part succeeds then we attempt the try-part of the alternative. If the try-part is successful, then we are done; if not, then we must look for another alternative. The backtracking involved is well-behaved in the sense that we backtrack only over entire alternatives, never over individual subgoals in the try-part of an alternative. All that must be undone is the bindings made in accomplishing the unification of the actual parameter list with the argument list in the match-part of the alternative.

We impose determinism on the program at this stage by insisting on a particular order for considering the alternatives: the order in which the user supplied them to the system. An extension to the system is desired that would analyze the alternatives and decide for itself what order would be most effective. The ordering of the subgoals is also open to question. These issues are being investigated by M. Franusich, a student at UCSC, and we hope to incorporate his results eventually.

The intermediate form of a generic specification involves two things. First, the intermediate form of a generic definition contains only the parameter list of the function and a list associating input-patterns with the function name to be used when that input-pattern is recognized. Although it does not appear in the internal generic definition, a generic specification causes the function definitions for each alternative version of the function to be made.

The Intermediate Language is similar to the Input Language in many ways. To point out the similarities, we have used the same nonterminal names where applicable. All the nonterminals are prefixed with a "$" to distinguish them from those of the Input Language.

5.1 Syntax of Intermediate Language

A context-free grammar describing the Intermediate Language is:

PHRASE STRUCTURE:

program ::= *$definition**

$definition ::= *$fun-def* | *$type-def* | *$gen-def* | *$target-def*

$fun-def ::= "(" "*function*" *$name $input-pattern $id-list*
$precondition $postcondition $body ")"

$type-def ::= "(" "*type*" *$name* "*(1 0)*" "*(x y)*" "*T*" "*(boolean y)*" *$body* ")"

$gen-def ::= "(" "*generic*" *$name $id-list $selection** ")"

$target-def ::= "(" "*setq*" "*target*" *$target-language* ")"

$target-language ::= *$name*

$selection ::= *$input-pattern $name*

$input-pattern ::= "(" *$zero-or-one** ")"

$zero-or-one ::= "*0*" | "*1*"

$name ::= *$identifier*

$id-list ::= "(" *name** ")"

$precondition ::= *$disjunction*

$postcondition ::= *$disjunction*

$disjunction ::= *$conjunction* | "(" "∨" *$conjunction $disjunction* ")"

$conjunction ::= *$literal* | "(" "∧" *$literal $conjunction* ")"

$literal ::= "*T*" | *$pred-app* | *$disjunction*

$pred-app ::= "(" *$name $arg** ")"

$body ::= *$backtracking-conditional*

$backtracking-conditional ::= "(" "bktrkcond" *$alternatives* ")"

$alternatives ::= *$match-try-pair**

$match-try-pair ::= "(" *$arglist* "(" "try" *$subgoals* ")" ")"

$arglist ::= "(" *$arg** ")"

$arg ::= *$fun-app* | *$variable* | *$constant*

$fun-app ::= "(" *$name $arg** ")"

$subgoals ::= *$pred-app**

$variable ::= *identifier*

$constant ::= *number* | *$string* | "*true*" | "*false*" | "*undef*" | "(" ")" | *$quoted-const*

$string ::= "(" "*string*" *$char-list* ")"

$char-list ::= "(" [*anychar* | *$punctuation* | *dblquote* | " "]* ")"

$punctuation ::= "," | "/." | "'" | "/'" | ";" | ":" | "//" | "/]" | "/(" | "/)"

$quoted-const ::= "(" "*quote*" *$exp* ")"

$exp ::= *identifier* | "(" *$elem** ")"

$elem ::= *$constant* | *$variable* | *$exp*

The LEXICON is the same as for the specification language.

5.2 Semantics of the Intermediate Language

The semantics of the intermediate language form of a function definition:

$fun-def ::= "(" "*function*" *$name $input-pattern $id-list $precondition*
$postcondition $body ")"

is given by the first order logic formula:

$$\forall \bar{x}\ \text{defined}[\ \text{inputs}[S[\$id\text{-}list], S[\$input\text{-}pattern]]\ ] \wedge S[\$precondition] \rightarrow S[\$body] \wedge (S[\$name\ \$id\text{-}list] \rightarrow S[\$postcondition])$$

where:

the semantic function S maps elements of the intermediate language onto their denotations in first-order logic. This is the obvious mapping for *$id-list*, *$input-pattern*, *$name*, *$precondition*, and *$postcondition*. (see page 25

$\bar{x}$ represents all variables in S[*$id-list*]

inputs[S[*$id-list*], S[*$input-pattern*]] is the list of input parameters, i.e.,
those parameters in S[*$id-list*] corresponding to 1's in S[*$input-pattern*]
defined[l] is true iff every element of the list l has a value that is
completely defined (contains no free variables)
and the semantics of the function body S[*$body*] is given below.

The intermediate form of a function body is:

"(" "*bktrkcond*" "(" $*arglist*$_1$ "(" "*try*" $*subgoals*$_1$ ")" ")"
...
"(" $*arglist*$_n$ "(" "*try*" $*subgoals*$_n$ ")" ")" ")"

Informally, the semantics can be expressed as a conjunction of implications each of which states that if the values bound to the *$id-list* are an instantiation of $*arglist*$_i$ then the conjunction of all the subgoals in $*subgoals*$_i$ implies (*$name $arglist*$_i$). Of course any substitutions necessary to unify the *$id-list* and $*arglist*$_i$ must be made throughout the $*subgoals*$_i$.

We actually want an "ordered" conjunction, to ensure that the implications involved in each alternative are considered in the order given. We want to say that if both antecedents in the implication are true then we can successfully claim the consequent and do not wish to consider the remaining implications as applicable; however, if either antecedent is false, then we want to proceed looking for an implication that is "useful" to us.

Let us represent by σ_i the substitution applied to unify [6] $*arglist*$_i$ with *$id-list*, if such a unification is possible. For all A, Aσ_i is A with all substitutions in σ_i made.

---

[6]Unification is the process of finding a "most general substitution", in the sense of making as few bindings as possible, which will render the objects being unified to be syntactically identical. See [D 76], or [M 64] for a complete definition.

Expressed as a formula of first order logic:

$$\bigwedge_{i=1}^{n} \; [\; (\text{unify}[S[\$id\text{-}list],\, S[\$arglist_i]] = \sigma_i) \;\wedge\; \bigwedge_{j=1}^{i-1} (\neg b_j) \;\rightarrow\; (S[\$subgoals_i\sigma_i] \rightarrow S[\$name\ \$arglist_i]\sigma_i) \;]$$

where:

S is again obvious on *$arglist*'s and *$subgoals*'s

b_j stands for $(\text{unify}[S[\$id\text{-}list],\, S[\$arglist_j]] = \sigma_j) \wedge S[\$subgoals_j\sigma_j]$

$S[\$subgoals_i\sigma_i]$ is the conjunction of all subgoals in $\$subgoals_i$ with the substitution σ_i made throughout,

and all free variables are universally quantified.

The semantics of a type definition is, again, very similar to that of a function definition. The semantics of the type definition:

"(" "*type*" *$name* "*(1 0)*" "*(x y)*" "*T*" "*(boolean y)*" *$body* ")"

is given by the formula:

$$\forall x \;\; \text{defined}[\,(x)\,] \;\rightarrow\; S[\$body] \wedge ((S[\$name\ (x\ y)]) \rightarrow S[(boolean\ y)])$$

where S[*$body*] is as defined above for function bodies.

The semantics of a generic definition:

"(" "*generic*" *$name $id-list*
*$input-pattern*$_1$ *$name*$_1$
...
*$input-pattern*$_n$ *$name*$_n$ ")"

is given through an association of a call on *$name* with the $\$name_i$ determined by the positions of the arguments having values at the time the function is called. Formally:

$$\bigwedge_{i=1}^{n} \; \forall x \; [\; S[\text{def-of}[\$name_i]] \wedge (S[\$name_i\ \$id\text{-}list] \rightarrow S[\$name\ \$id\text{-}list]) \;]$$

where: S[def-of[$\$name_i$]] is the semantics of the function definition of $\$name_i$.
i.e.,

$$\forall \bar{x} \; \text{defined}[\; \text{inputs}[S[\$id\text{-}list],\, S[\$input\text{-}pattern_i]] \;] \wedge S[\$precondition_i] \rightarrow$$

$$S[\text{body-of}[\$name_i]] \wedge (S[\$name_i\ \$id\text{-}list] \rightarrow S[\$postcondition_i])$$

The semantics of a target definition:

"(" "*setq*" "*target*" $name ")"

is the Hoare rule for assignment:

$$P[\$name/target]\ \{\ (setq\ target\ \$name)\ \}\ P$$

Mapping From Input Specification to Intermediate Form

We now define the mapping *I* (for "internalize") from input specifications into intermediate language. In the first column we identify the input form being translated, the center column shows the translation, and the third column relates nonterminals of the input language to the instance being translated. Since the lexicon for both languages is the same, the translation is concerned only with the phrase structure of the languages.

$I[input] =$	$I[definition^*]$	$input = definition^*$
$I[\epsilon] =$	ϵ	where ϵ is the empty string
$I[definition_1\ definition^*] =$	$I[definition_1]\ I[definition^*]$	
$I[definition] =$	$I[fun\text{-}def]$	$definition = fun\text{-}def$
$I[definition] =$	$I[type\text{-}def]$	$definition = type\text{-}def$

I[definition] =	*I[gen-def]*	*definition* = *gen-def*
I[definition] =	*I[target-def]*	*definition* = *target-def*
I[fun-def] =	"(*function*" *name input-pattern id-list I[precondition] I[postcondition]* "(" "*bktrkcond*" *I[h-clause**] ")" ")"*	*fun-def*="*function*" *name* "*input pattern?*" *input-pattern* "*parameter list?*" *id-list* "*precond?*" *precondition* "*postcond?*" *postcondition* "*body?*" *h-clause** "."

The function *I* is the identity mapping on *name*'s, *id-list*'s, and *input-pattern*'s, and on *precondition*'s and *postcondition*'s it is simply a translation to a fully parenthesized prefix form of representation.

I[precondition] =	*I[disjunction]*	*precondition* = *disjunction* "."
I[postcondition] =	*I[disjunction]*	*postcondition* = *disjunction* "."
I[disjunction] =	*I[conjunction]*	*disjunction* = *conjunction*
I[disjunction] =	"(∨" *I[conjunction] I[disjunction]* ")"	*disjunction* = *conjunction* "∨" *disjunction*
I[conjunction] =	*I[literal]*	*conjunction* = *literal*
I[conjunction] =	"(∧" *I[literal] I[conjunction]* ")"	*conjunction* = *literal* "∧" *conjunction*
I[literal] =	"*T*"	*literal* = "*TRUE*"
I[literal] =	"*T*"	*literal* = "*T*"
I[literal] =	*I[pred-app]*	*literal* = *pred-app*
I[literal] =	*I[disjunction]*	*literal* = "(" *disjunction* ")"
I[pred-app] =	*I[name arglist]*	*pred-app* = *fname*[7] *arglist*

[7]The internalized form of a generic application will have a different name

I[name arglist] =	"(" *name I[arg*]* ")"	*arglist* = "(" *arg** ")"
I[arg$_1$ arg]* =	*I[arg$_1$] I[arg*]*	
I[arg] =	*I[constant]*	*arg* = *constant*
I[arg] =	*variable*	*arg* = *variable*
I[arg] =	*I[fun-app]*	*arg* = *fun-app*
I[constant] =	*number*	*constant* = *number*
I[constant] =	"(" "*string*" *I[string]* ")"	*constant* = *string*
I[constant] =	"*true*"	*constant* = "*true*"
I[constant] =	"*true*"	*constant* = "*t*"
I[constant] =	"*false*"	*constant* = "*false*"
I[constant] =	"*false*"	*constant* = "*f*"
I[constant] =	"*undef*"	*constant* = "*undef*"
I[constant] =	"(" ")"	*constant* = "(" ")"
I[constant] =	*I[quoted-const]*	*constant* = *quoted-const*
I[fun-app] =	*I[name arglist]*	*fun-app* = *name arglist*
I[string] =	"(" "*string*" *I[* [*anychar* | *punctuation* | *dblquote dblquote* | " "]*] ")"	*string* = *dblquote* [*anychar* | *punctuation* | *dblquote dblquote* | " "]* *dblquote*

I[[*anychar* | *punctuation*
| *dblquote dblquote* | " "]*] =
"(" [*I[anychar]*
| *I[punctuation]*

---

substituted for the name of the generic if it appears in the specification of the body of a function. This is why we listed *name* as the translate of *fname* above. The name which is chosen will depend on the input pattern of the function being specified.

| /[dblquote dblquote]
| /[" "]]* ")"

/[anychar] =	anychar	
/[punctuation] =	punctuation	punctuation = ";" | ":" | "(" | ")" | "," | "."
/[dblquote dblquote] =	dblquote	
/[" "] =	"/ "	
/[quoted-const] =	"(" "quote" /[exp] ")"	quoted-const = "'" exp
/[exp] =	identifier	exp = identifier
/[exp] =	"(" /[elem*] ")"	exp = "(" elem* ")"
/[elem$_1$ elem*] =	/[elem$_1$] /[elem*]	
/[elem] =	/[constant]	elem = constant
/[elem] =	/[variable]	elem = variable
/[elem] =	/[exp]	elem = exp
/[variable] =	identifier	variable = identifier
/[variable] =	!identifier	variable = identifier formal parameters are indicated by a first character "!"

The interesting part of / maps Horn clauses onto the intermediate form of the function body. Whenever an instance of a generic call is encountered in the body of a specification, the first version of the generic whose input pattern is satisfied, i.e. all argument positions corresponding to 1's in the input pattern have values supplied, is

substituted for the generic. Thus we never have to translate generic specifications into the target language; we just translate each specific version.

I[h-clause$_1$ h-clause*] = I[h-clause$_1$] I[h-clause*]

I[h-clause] =	*I[goal]*	*h-clause = goal*
I[goal] =	*"(" arglist "(" "try" ")" ")"*	*goal = pred-app* *= name arglist*
I[h-clause] =	*"(" arglist* *"(" "try" I[subgoals] ")" ")"*	*h-clause = goal "←" subgoals* *goal = name arglist*
I[subgoals] =	*I[pred-app$_1$]* *I[["," pred-app]*]*	*subgoals =* *pred-app$_1$ ["," pred-app]**

I["," pred-app$_1$
["," pred-app]*] = I[pred-app$_1$] I[["," pred-app]*]

For type specifications we have:

I[type-def] =	*"(" "type" name "(1 0)"* *"(x y)" "T" "(boolean y t)"* *"(" "bktrkcond" I[h-clause*] ")"*	*type-def =* *"type" name* *"body" h-clause* "."*

As discussed earlier, one may consider the type definition to be the definition of a predicate to recognize instances of the type. We must make a distinction between a type definition and the recognizer for a type when our target language is strongly typed. This will be discussed in the mapping from intermediate form to target language.

The mapping from the input specification of a generic function to its intermediate form involves not only the internal form of the generic definition but

also that of each of the specific versions of the generic being defined. The function **Select** creates a sequence of *input-pattern fname* pairs from which the choices of individual functions to substitute for generic calls will be made. The function **Make-defs** creates a function definition for each version of the generic. The resulting definitions are identical to those that would have been created if each version were specified explicitly as a *fun-def*.[8]

I[*gen-def*][9] = "(" "*generic*" *gname id-list*
Select[[*choice* "*choices?*"]*] ")"
Make-defs[*id-list*,
[*choice* "*choices?*"]*
*bodydef**]

gen-def =
"*generic*" *gname*
"*parameter list?*" *id-list*
"*choices?*" [*choice* "*choices?*"]* "."
"*body-defs:*" *body-def**

where:

Select[ϵ] = ϵ where ϵ is the empty string

Select[*choice*$_1$ "*choices?*" [*choice* "*choices?*"]*] =
Select[*choice*$_1$] **Select**[[*choice* "*choices?*"]*]

Select[*choice*] = *input-pattern fname*

choice =
input-pattern
"*function name?*" *fname*
"*precond?*" *precondition*
"*postcond?*" *postcondition*
"*body name?*" *bname*

Make-defs[*id-list*,[*choice* "*choices?*"]*, ϵ] = ϵ

Make-defs[*id-list*,ϵ, *body-def**] = ϵ

Make-defs[*id-list*, *choice*$_1$ "*choices?*" [*choice* "*choices?*"]*, *body-def**] =

---

[8]The variables *gname*, *fname*, and *bname*, in the following, are all instances of the nonterminal *name*.

[9]*I*[*gen-def*] is of the form *$gendef $fundef** in the intermediate language.

Make-def[*id-list*, *choice*$_1$, **Body-of**[*choice*$_1$, *body-def**]]
Make-defs[*id-list*, [*choice* "*choices?*"]*, *body-def**]

Body-of[*choice*, *body-def**] = the *body-def* whose *bname* is mentioned in *choice*.

Make-def[*id-list*, *choice*, *body-def*] =
"(" "*function*" *fname*
input-pattern id-list precondition
postcondition [*h-clause**] ")"

choice =
input-pattern
"*function name?*" *fname*
"*precond?*" *precondition*
"*postcond?*" *postcondition*
"*body name?*" *bname*
body-def =
bname h-clause* "."

Finally,

[*target-def*] = "(" "*setq*" "*target*"
target-language ")"

target-def =
"*target*" *target-language*

The target language specification is just information to the system indicating which target language is desired and is of interest only in choosing the mapping from intermediate language to target.

The specification of the factorial function:

function Fact
input pattern? (*1 0*)
parameter list? (*x y*)
precond? Integer(*x*, *true*) ∧ ≥(*x*, *0*, *true*).
postcond? Integer(*y*, *true*) ∧ >(*y*, *0*, *true*).
body? Fact(*0*, *1*)
Fact(*z*, *w*) ← *Subl*(*z*, *z1*), *Fact*(*z1*, *w1*), ***(*z*, *w1*, *w*).

has the following form in the Intermediate Language:

```
(function Fact (1 0) (!x !y)
      (∧ (Integer !x true) (≥ !x 0 true))
      (∧ (Integer !y true) (> !y 0 true))
      (bktrkcond ((0 1) (try))
            ((!z !w) (try (Sub! !x !z1)
                          (Fact !z1 !w1)
                          (* !z !w1 !w) ) )
      ) )
```

Correctness of the Mapping to Intermediate Form

We wish to show that the semantics of an input specification is "sufficiently" preserved in the mapping to intermediate form. By "sufficiently" we mean that the semantics of the intermediate form of a function definition is implied by the semantics of the input specification, and furthermore, if every function (or type) P is defined such that for each input tuple, $\bar{x}_i$, (i.e., tuple with which the function is called) there exists a unique output tuple, $\bar{x}_o$, such that $P(\bar{x}_o)$ is derivable using the input semantics, then $P(\bar{x}_o)$ is derivable using the semantics of the intermediate form of the definition. We refer to this restriction of requiring uniqueness of the output tuple by saying that we are defining only "functional" relationships.

The semantics of input and internal form are not equivalent because the input semantics may be used several ways to generate proofs. This is due to the

nondeterministic nature of the process of finding proofs. At any stage of a proof, there may be many Horn clauses that are applicable as axioms, and different proof paths may generate different results. However, we have restricted ourselves to defining functional relationships rather than the more general "relations" which may be one-to-many mappings. Thus, no matter what direction the proof takes, there is a unique result. If a function is specified that does not return unique answers, then the resulting program (and any others whose specifications make use of the function) will be partially correct; however we cannot guarantee that any specification using such a function will result in a program that always finds an answer when one exists.

In the intermediate form we are forced to apply the Horn-clauses in a specific order. Due to the determinism introduced we will follow precisely one proof, the same proof, every time a procedure gets called with the same input values. The important point to establish is that whenever a unique answer may be found in the nondeterministic proof process, our deterministic search will find it as well. First, we establish two lemmas.

Lemma 1.1

$$\forall \bar{x}\ S[h\text{-}clause^*] \rightarrow S[I[h\text{-}clause^*]]$$

that is

$$\forall \bar{x}\ [\ \bigwedge_{i=1}^{n}\ (\text{name arglist}_i \leftarrow S[subgoals_i]) \rightarrow \bigwedge_{i=1}^{n} (\text{unify}[\text{id-list}, \text{arglist}_i] = \sigma_i) \wedge \bigwedge_{j=1}^{i-1} \neg(S[I[subgoals_j]\sigma_j]) \rightarrow (S[I[subgoals_i]\sigma_i] \rightarrow S[I[name\ arglist_i]\sigma_i])\ ]$$

where $\forall \bar{x}$ stands for the universal quantification of all variables mentioned in the quantified formula.

Proof:

We shall show that for each i, $S[h\text{-}clause] \rightarrow S[I[h\text{-}clause]]$. For the sake of readability we shall leave out the quantifier with the understanding that all variables are universally quantified. Thus, we want to show:

$$(\text{name arglist}_i \leftarrow S[subgoals_i]) \rightarrow \\ ((\text{unify[id-list, arglist}_i] = \sigma_i) \wedge \\ \bigwedge_{j=1}^{i-1} \neg(S[I[subgoals_j]\sigma_j]) \rightarrow \\ (S[I[subgoals_i]\sigma_i] \rightarrow S[I[name\ arglist_i]\sigma_i]))$$

1. $S[subgoals_i] \rightarrow \text{name arglist}_i$	hypothesis
2. $[S[subgoals_i] \rightarrow \text{name arglist}_i]\sigma_i$	$\forall x\ A(x) \vdash A(t)$ (as many times as necessary, given that we change variable names that are introduced by the substitution in order to ensure that the new variables are *free for* [10] the old.
3. $[S[I[subgoals_i]] \rightarrow S[I[name\ arglist_i]]]\sigma_i$	from 2., equivalent replacement, and the facts that $S[I[subgoals_i]] = S[subgoals_i]$ and $S[I[name\ arglist_i]] = S[name\ arglist_i]$
4. $S[I[subgoals_i]\sigma_i] \rightarrow S[I[name\ arglist_i]]\sigma_i$	property of substitution
5. $(\text{unify[id-list, arglist}_i] = \sigma_i) \wedge \bigwedge_{j=1}^{i-1} \neg S[I[subgoals_j]\sigma_j] \rightarrow (S[I[subgoals_i]\sigma_i] \rightarrow S[I[name\ arglist_i]]\sigma_i)$	$A \rightarrow (B \rightarrow A)$ and 4., and Modus Ponens
6. $(\text{name arglist}_i \leftarrow S[subgoals_i]) \rightarrow ((\text{unify[id-list, arglist}_i] = \sigma_i) \wedge \bigwedge_{j=1}^{i-1} \neg S[I[subgoals_j]\sigma_j] \rightarrow (S[I[subgoals_i]\sigma_i] \rightarrow S[I[name\ arglist_i]\sigma_i]))$	Deduction Theorem, 1.,5.

QED

---

[10] For a formal definition of *free for* see Davis [D 76], Kleene [K 52], or Mendelson [M 64].

The following lemma makes use of two sub-lemmas whose proofs follow it.

Lemma 1.2

$$\forall\bar{x}\ (Y \rightarrow Y') \vdash \forall\bar{x}(X \rightarrow Y \wedge Z) \rightarrow \forall\bar{x}(X \rightarrow Y' \wedge Z)$$

where X, Y, Y',and Z are any well-formed-formulas. The proof is given for a single quantified variable. Clearly it can be generalized to any number of variables.

1. $\forall x(X \rightarrow Y \wedge Z)$	hypothesis
2. $\forall x(X \rightarrow Y)$	1., Lemma 1.2.1
3. $X \rightarrow Y$	2., ∀-elimination
4. $\forall x(Y \rightarrow Y')$	hypothesis
5. $Y \rightarrow Y'$	4., ∀-elimination
6. $X \rightarrow Y'$	3.,5., transitivity of →
7. $\forall x(X \rightarrow Y')$	6., ∀-introduction
8. $\forall x(X \rightarrow Z)$	1., Lemma 1.2.1
9. $\forall x(X \rightarrow Y' \wedge Z)$	7.,8., Lemma 1.2.2
9. $\forall x(X \rightarrow Y \wedge Z) \rightarrow \forall x(X \rightarrow Y' \wedge Z)$	1.,9., Deduction Theorem

Lemma 1.2.1

$$\forall x(X \rightarrow Y \wedge Z) \vdash \forall x(X \rightarrow Y)$$

we will derive the above version, and by commutativity of ∧ assume that we also have

$$\forall x(X \rightarrow Y \wedge Z) \vdash \forall x(X \rightarrow Z)$$

1. $\forall x(X \rightarrow Y \wedge Z)$	hypothesis
2. $X \rightarrow Y \wedge Z$	∀-elimination
3. X	hypothesis
4. $Y \wedge Z$	2.,3., Modus Ponens
5. Y	∧-elimination
6. $X \rightarrow Y$	3.,5., Deduction Theorem
7. $\forall x(X \rightarrow Y)$	6., ∀-introduction

Lemma 1.2.2

$$\forall x(X \rightarrow Y), \forall x(X \rightarrow Z) \vdash \forall x(X \rightarrow Y \wedge Z)$$

1. $\forall x(X \rightarrow Y)$	hypothesis
2. $X \rightarrow Y$	1., $\forall$-elimination
3. $\forall x(X \rightarrow Z)$	hypothesis
4. $X \rightarrow Z$	3., $\forall$-elimination
5. X	hypothesis
6. Y	2.,5., Modus Ponens
7. Z	4.,5., Modus Ponens
8. $Y \wedge Z$	$\wedge$-introduction
9. $X \rightarrow Y \wedge Z$	5.,8., Deduction Theorem
10. $\forall x(X \rightarrow Y \wedge Z)$	$\forall$-introduction

We are now in a position to prove our first theorem.

Theorem 1 :

$$S[I[input]] \vdash P(\bar{x}) \Rightarrow S[input] \vdash P(\bar{x})$$

Any proof derived from the semantics of the intermediate form could be derived from the semantics of the input specification. We will show this by determining that the semantics of an input specification implies the semantics of the intermediate form of the specification.

Proof:

Since *input* is a sequence of definitions which gets internalized as a sequence of definitions, we shall show that:

$$S[definition] \rightarrow S[I[definition]]$$

Then the conjunction of the input definitions implies the conjunction of the intermediate language form of those definitions and:

$$S[input] \rightarrow S[I[input]]$$

and thus anything derivable from $S[I[input]]$ can be derived from $S[input]$ by first deriving $S[I[input]]$ and then following the same proof. We must establish two things: [11]

---

[11] Recall that *I* is just the identity mapping on *id-list*'s, *input-pattern*'s, and *name*'s.

1) for function and type definitions [12]:

$$[\ \forall\bar{x}\ \text{defined}[\ \text{inputs}[S[\textit{id-list}], S[\textit{input-pattern}]]\ ] \wedge S[\textit{precondition}] \rightarrow S[\textit{h-clause}^*] \wedge (\ (S[\textit{name}]\ S[\textit{id-list}]) \rightarrow S[\textit{postcondition}])\ ]$$
$$\rightarrow$$
$$[\ \forall\bar{x}\ \text{defined}[\ \text{inputs}[S[\textit{id-list}], S[\textit{input-pattern}]]\ ] \wedge S[\textit{I}[\textit{precondition}] \rightarrow S[\textit{I}[\textit{h-clause}^*]] \wedge (\ (S[\textit{name}]\ S[\textit{id-list}]) \rightarrow S[\textit{I}[\textit{postcondition}])\ ]$$

2) for generic definitions:

$$\bigwedge_{i=1}^{n}\ \forall\bar{x}\ [\ (S[\textit{funname}_i\ \textit{id-list}] \rightarrow S[\textit{name id-list}]) \wedge (\ \text{defined}[\ \text{inputs}[S[\textit{id-list}], S[\textit{input-pattern}_i]]\ ] \wedge S[\textit{precondition}_i] \rightarrow S[\text{H-C}[\textit{body-name}_i]] \wedge (S[\textit{funname}_i\ \textit{id-list}] \rightarrow S[\textit{postcondition}_i])\ )\ ]$$
$$\rightarrow$$
$$\bigwedge_{i=1}^{n}\ \forall\bar{x}\ [\ (S[\textit{funname}_i\ \textit{id-list}] \rightarrow S[\textit{name id-list}]) \wedge (\ \text{defined}[\ \text{inputs}[S[\textit{id-list}], S[\textit{input-pattern}_i]]\ ] \wedge S[\textit{I}[\textit{precondition}_i]] \rightarrow S[\textit{I}[\text{H-C}[\textit{body-name}_i]]] \wedge (S[\textit{funname}_i\ \textit{id-list}] \rightarrow S[\textit{I}[\textit{postcondition}_i]])\ )\ ]$$

where H-C[$\textit{body-name}_i$] is the $\textit{h-clause}^*$ associated with $\textit{body-name}_i$.

1) is of the form $\forall\bar{x}\ (A \wedge B \rightarrow C \wedge D) \rightarrow \forall\bar{x}\ (A \wedge B \rightarrow C' \wedge D)$, where we know by Lemma 1.1 that $\forall\bar{x}(C \rightarrow C')$. Thus, this is just an instance of Lemma 1.2 with "$A \wedge B$" for X, "C" for Y, "C'" for Y', and "D" for Z.

2) for each i, is of the form $\forall\bar{x}[(A \rightarrow B) \wedge (C \wedge D \rightarrow E \wedge F)] \rightarrow \forall\bar{x}[(A \rightarrow B) \wedge (C \wedge D \rightarrow E' \wedge F)]$. This is just $X \wedge Y \rightarrow X \wedge Y'$, where we know by part 1) that $Y \rightarrow Y'$. For each i, the statement is true, thus the conjunction of statements over all i's is true.

QED

---

[12] The semantics of a type definition is the same as that for a function definition. We gave them separately before, pointing out that we know what the id-list, input-pattern, precondition, and postcondition are for type definitions.

Theorem 2:

If each definition is functional with respect to its output variables, that is, if for each function or type P and each input tuple $\bar{x}_i$, there exists a unique output tuple, $\bar{x}_o$, such that $P(\bar{x}_o)$, then:

$$S[input] \vdash P(\bar{x}_o) \Rightarrow S[I[input]] \vdash P(\bar{x}_o)$$

Proof:

Given a proof of $P(\bar{x})$ from S[*input*] we can construct a proof from S[*I*[*input*]]. A clause of the definition of P is applicable if and only if the current goal is an instance of the head of the clause, i.e., there is a σ_i such that unify[id-list, arglist$_i$] = σ_i where id-list is bound to the actual parameters in the goal. Thus the only time when a clause that is applicable in the proof from S[*input*] may not be used in a proof from S[*I*[*input*]] is when there is a previously listed clause that is also applicable.

Suppose the proof of $P(\bar{x})$ makes use of the clause P_i of the definition of P when there is a clause P_j, j<i, that is also applicable. Our proof from S[*I*[*input*]] will attempt to prove $P(\bar{x})$ using P_j. We are guaranteed termination of this attempt by the precondition. If we terminate successfully, we will have proved $P(\bar{x})$, since this proof may also be considered a proof from S[*input*] and we have assumed that $\bar{x}$ is unique. If we terminate unsuccessfully, then we will attempt another clause, possibly P_i now, as in the original proof from S[*input*], or possibly some P_k, j<k<i. We will eventually succeed before attempting P_j, or we will use P_i as we did

in the original proof. The same analysis is applicable to every clause used in the proof of $P(\bar{x})$ from S[*input*] and thus we can derive $P(\bar{x})$ from S[*I*[*input*]].

QED

Mapping the Intermediate Language to LISP

We have chosen to use LISP as the first target language. One may find documentation for the version we use, MACLISP, in [M 74]. We generate "pure LISP" programs, using none of the special features of the MACLISP implementation, however the names of the primitive functions may vary from implementation to implementation. These functions include: *defun*, for defining a function, *putprop*, for putting a property on the property list of an atom, and *get*, for getting a property from the property list. The function *L*, mapping the intermediate language into LISP is defined as follows.

$L[\epsilon] = \epsilon$

$L[\$definition_1\ \$definition^*] = L[\$definition_1]\ L[\$definition^*]$

$L[\$definition] = L[\$fun\text{-}def] \quad \$definition = \$fun\text{-}def$

$L[\$definition] = L[\$type\text{-}def] \quad \$definition = \$type\text{-}def$

```
L[$fun-def] = "(putprop" "'"$name                    $fun-def =
                "'"$input-pattern "'inpat)"          "(" "function" $name
              "(putprop" "'"$name "'"$id-list         $input-pattern
                    "'params)"                       $id-list $precondition
              "(putprop" "'"$name "'"$precondition   $postcondition "("
                    "'precond)"                      "bktrkcond" $alternatives ")" ")"
              "(putprop" "'"$name "'"$postcondition
                    "'postcond )"
              "(defun" $name "fexpr (l)"
                    "(bktrkcond l" "'"$alternatives"))"

L[type-def] = "(putprop" "'"$name                    type-def =
                "'(1 0) 'inpat)"                     "(" "type" $name
              "(putprop "'"$name "'(x y)"            "(1 0)" "(x y)"
                    "'params)"                       "T" "(boolean y)"
              "(putprop "'"$name                     "(" "bktrkcond"
                "'T 'precond)"                       $alternatives ")" ")"
              "(putprop "'"$name
                "'(boolean y) 'postcond)"
              "(defun" $name "fexpr (l)"
                    "(bktrkcond l "'"$alternatives"))"
```

There is no mapping of generic definitions, only of the specific function definitions involved, and these are identical to that for *fun-def*'s above. The LISP function *bktrkcond* evaluates its arguments, binds the results to the formal parameters *actuals* and *list-alts*, and then recursively attempts each alternative until an answer is found or all alternatives have failed, indicating that the answer is undefined. The complete set of definitions of LISP functions that implement the mapping, i.e. definitions for *bktrkcond*, *match*, *try*, and all of the subfunctions required, may be found in Section 16.1, beginning at page 162.

The LISP program that is generated from the specification for the factorial function is as follows. The list of *putprop*'s get evaluated only once; they represent global information about the properties of the function. The *defun* is the actual LISP definition of the function.

```
(defun Fact fexpr (l)
    (cond ((true-precond (cons 'Fact l))
           (bktrkcond l
                      '(((0 1) (try))
                        ((!x !y)
                         (try (Subl !x !x1)
                              (Fact !x1 !y1)
                              (* !x !y1 !y)))))))
          (t 'undef)))
```

Correctness of the Mapping to LISP Program

We shall show that the semantics of the LISP form of a function or type definition is equivalent to the semantics of the intermediate form of a function or type definition. There is no distinction between function and type definitions in either form.

We have shown earlier that the semantics of the intermediate form of a definition is expressed:

$$
\begin{aligned}
&\forall \bar{x}\ \text{defined}[\ \text{inputs}[\$\text{id-list}, \$\text{input-pattern}]\ ] \wedge \$\text{precondition} \rightarrow \\
&\bigwedge_{j=1}^{n} [\\
&\quad (\text{unify}[\$\text{id-list}, \$\text{arglist}_j] = \sigma_j) \\
&\quad \wedge \bigwedge_{i=1}^{j-1} (\text{unify}[\$\text{id-list}, \$\text{arglist}_i] = \sigma_i \rightarrow \neg S[\$\mathit{subgoals}_i\sigma_i]) \\
&\quad \rightarrow \ (\ S[\$\mathit{subgoals}_j\sigma_j] \rightarrow (\$\text{name}\ \$\text{arglist}_j)\sigma_j\ )\ ] \\
&\wedge \\
&\forall \bar{x}\ \$\text{name}\ \$\text{id-list} \rightarrow \$\text{postcondition}
\end{aligned}
$$

Since $arglist$_i$ and $id-list are unified by σ_i, we can use $name $id-list instead of $name $arglist$_i$ in the final consequent of the first conjunct above. Another way of expressing the fact that $name $id-list is true is by saying that $id-list ϵ $name, that is, the given tuple is an element of the relation $name. Before turning our attention to the semantics of the LISP form, we shall rewrite the semantics of the intermediate form in such a way as to facilitate our proof. This rewriting is based on several rules of first-order logic. To make the statements more readable, we shall look at the general form of the rewrite first, and then apply the result to the expression above.

We are going to focus our attention on the first conjunct of the above expression. This is of the form:

$$\forall \bar{x}\ A \rightarrow \bigwedge_{i=1}^{n} [B \wedge C \rightarrow (D \rightarrow E)]$$

which is equivalent to

$$\forall \bar{x}\ A \rightarrow \forall i\ [B \wedge C \rightarrow (D \rightarrow E)]$$

where actually the quantifier on i is bounded. Since we know there are a finite number of i's we will not put in the bounds. It should be understood that i ranges over the number of clauses given in the definition. [13] The above statement is in turn equivalent to

$$\forall \bar{x}\ A \rightarrow \forall i\ [(B \wedge C \wedge D) \rightarrow E]$$

Since in our original formula E does not contain any free occurrences of i, this is equivalent to:

$$\forall \bar{x}\ A \rightarrow [\exists i (B \wedge C \wedge D) \rightarrow E]$$

---

[13] The "variable" i is not really a variable at all in the formal sense; it is merely an index. We could write out the consequence of the above implication as the conjunction of the indexed statements. We are using the "$\forall i$" notation because it provides a convenient abbreviation.

Thus our new formulation of the semantics of the intermediate form of a function (or type) definition is:

$$\forall \bar{x}\ \text{defined}[\ \text{inputs}[\$\text{id-list}, \$\text{input-pattern}]\ ] \wedge \$\text{precondition} \rightarrow$$
$$[\ \exists i\ (\ \text{unify}[\$\text{id-list}, \$\text{arglist}_i] = \sigma_i$$
$$\wedge\ \bigwedge_{j=1}^{i-1} (\text{unify}[\$\text{id-list}, \$\text{arglist}_j] = \sigma_j \rightarrow \neg S[\$subgoals_j\sigma_j])$$
$$\wedge\ S[\$subgoals_i\sigma_i]\ )$$
$$\rightarrow\ (\$\text{id-list})\sigma_i \in \$\text{name})\ ]$$
$$\wedge$$
$$\forall \bar{x}\ (\$\text{id-list} \in \$\text{name}) \rightarrow \$\text{postcondition}$$

The semantics of a LISP form of a definition, such as:

```
(putprop $name $input-pattern 'inpat)
(putprop $name $id-list 'params)
(putprop $name $precondition 'precond)
(putprop $name $postcondition 'postcond)
(defun $name fexpr (l)
      (cond ((true-precond (cons $name l))
             (bktrkcond l (rest $body-def)))
            (t undef)))
```

is:

$$\forall \bar{x}\ [\ \text{eval}[(\text{cons } \$\text{name actuals})] = \sigma \wedge \sigma \neq \text{undef} \rightarrow (\text{actuals})\sigma \in \$\text{name}\ ]$$
$$\wedge\ \forall \bar{x}[\ (\text{actuals} \in \$\text{name}) \rightarrow \$\text{postcondition}]$$

where: σ is a substitution or σ = undef
actuals is an instantiation of *$id-list*
$body-def is (*bktrkcond* ($arglist$_1$ (*cons* 'try $subgoals$_1$))
($arglist$_2$ (*cons* 'try $subgoals$_2$))
...
($arglist$_n$ (*cons* 'try $subgoals$_n$)))

The actual function definition begins with *defun*, the preceding *putprop*'s simply attach the listed information to the property list of the function being defined. See [MT 79] for a formal axiomatization of the semantics of LISP. Using the definitions of eval and $name, we can write:

```
eval[(cons $name actuals)] =
      IF true-precond[(cons $name actuals)]
      THEN bktrkcond[actuals, rest[$body-def]]
```

ELSE undef

thus,

$$\text{eval}[(\text{cons } \$\text{name actuals})] = \sigma \wedge \sigma \neq \text{undef}$$
$$\text{is equivalent to}$$
$$\text{true-precond}[(\text{cons } \$\text{name actuals})] \wedge \text{bktrkcond}[\text{actuals}, \text{rest}[\$\text{body-def}]] = \sigma \wedge \sigma \neq \text{undef}$$

So we can rewrite the semantics of the LISP form as:

$$\forall \bar{x} \; [\; \text{true-precond}[(\text{cons } \$\text{name actuals})]$$
$$\wedge \; \text{bktrkcond}[\text{actuals}, ((\$\text{arglist}_1 \; (\text{cons 'try } \$\text{subgoals}_1))$$
$$(\$\text{arglist}_2 \; (\text{cons 'try } \$\text{subgoals}_2))$$
$$\ldots$$
$$(\$\text{arglist}_n \; (\text{cons 'try } \$\text{subgoals}_n)))] = \sigma$$
$$\wedge \; \sigma \neq \text{undef}$$
$$\rightarrow \; (\text{actuals})\sigma \in \$\text{name} \;]$$
$$\wedge \; \forall \bar{x} \; [(\text{actuals} \in \$\text{name}) \rightarrow \$\text{postcondition}]$$

We need to establish that the above statement is equivalent to the statement of the semantics of the intermediate form of definition given above. We intend to show this by first establishing that:

$$\forall \bar{x} \; [$$
$$[\text{defined}[\text{inputs}[\$\text{id-list}, \$\text{input-pattern}]] \wedge \$\text{precondition}] \rightarrow$$
$$[\; \exists i \; (\; \text{unify}[\$\text{id-list}, \$\text{arglist}_i] = \sigma_i$$
$$\wedge \; \bigwedge_{j=1}^{i-1} (\text{unify}[\$\text{id-list}, \$\text{arglist}_j] = \sigma_j \rightarrow \neg S[\$\mathit{subgoals}_j\sigma_j])$$
$$\wedge \; S[\$\mathit{subgoals}_i\sigma_i] \;)$$
$$\rightarrow \; (\$\text{id-list})\sigma_i \in \$\text{name}) \;]$$
$$=$$
$$[\; \text{true-precond}[(\text{cons } \$\text{name actuals})]$$
$$\wedge \; \text{bktrkcond}[\text{actuals}, ((\$\text{arglist}_1 \; (\text{cons 'try } \$\text{subgoals}_1))$$
$$(\$\text{arglist}_2 \; (\text{cons 'try } \$\text{subgoals}_2))$$
$$\ldots$$
$$(\$\text{arglist}_n \; (\text{cons 'try } \$\text{subgoals}_n)))] = \sigma$$
$$\wedge \; \sigma \neq \text{undef}$$
$$\rightarrow \; (\text{actuals})\sigma \in \$\text{name} \;] \;]$$

The expression resulting from distributing the $\forall \bar{x}$ over the equivalence follows easily from this stronger result. First of all, we will simplify the notation. Both $id-list in the

intermediate form and actuals in the Lisp form are names used to refer to instantiations of the formal parameter list. Since these are now within the scope of the same universal quantifier, we shall identify them both by the same name, actuals. We should also mention that precondition and postcondition are names that are being used to stand for the formulas they represent. Each involves some of the elements of the formal parameter list and we shall assume that there is no confusion as to the bindings of these variables even though we do not mention them explicitly in the formula.

Before attempting the main theorem of this section, we need four lemmas:

1. defined[inputs[actuals, $input-pattern]] ∧ $precondition
   ▪ true-precond[(cons $name actuals)]
2. unify[actuals, $arglist$_i$] ▪ σ_i
   ▪ match[actuals, newversion[$arglist$_i$]] ▪ $\sigma_i \wedge \sigma_i \neq$ undef
3. S[*$subgoals*] ▪ try[$subgoals] ▪ $\sigma \wedge \sigma \neq$ undef
4. ∃i (unify[$id-list, $arglist$_i$] ▪ σ_i
   ∧ $\bigwedge_{j=1}^{i-1}$ (unify[$id-list, $arglist$_j$] ▪ $\sigma_j \rightarrow \neg$S[*$subgoals$_j\sigma_j$*])
   ∧ S[*$subgoals$_i\sigma_i$*])

▪
bktrkcond[actuals, (($arglist$_1$ (cons 'try $subgoals$_1$))
($arglist$_2$ (cons 'try $subgoals$_2$))
...
($arglist$_n$ (cons 'try $subgoals$_n$))))] ▪ σ
∧ $\sigma \neq$ undef

We intend to show that the top level mapping is correct, however we do not intend to prove the entire implementation correct, so the proofs of these lemmas will be informal and assume correctness of several subfunctions involved.

Lemma 1

defined[inputs[actuals, $input-pattern]] ∧ $precondition
▪ true-precond[(cons $name actuals)]

Proof:

The function true-precond: 1) looks up the $input-pattern associated with $name and checks to see that all input positions of actuals have values supplied; then 2)looks up the $id-list and $precondition associated with $name, binds the variables in $id-list to the values supplied by actuals and evaluates the $precondition. Thus, true-precond returns "true" if and only if both steps1) and 2) are successful, i.e., if and only if defined[inputs[actuals, $input-pattern]] ∧ $precondition.

QED

Lemma 2

$$\text{unify[actuals, \$arglist}_i\text{]} = \sigma_i$$
$$\blacksquare \quad \text{match[actuals, newversion[\$arglist}_i\text{]]} = \sigma_i \wedge \sigma_i \neq \text{undef}$$

Proof:

The semantic function unify, attempts to unify its arguments after renaming variables so its arguments have no variable in common. If unification is possible, then unify returns a substitution, if not, then unify[actuals, $arglist$_i$] ≠ σ for any substitution σ, so the result is false.

newversion[$arglist] generates a new version of $arglist in which each variable has been replaced by a newly generated one. Thus, newversion ensures that the arguments to match have been standardized apart. match is a unification algorithm that returns a (most general) substitution σ if its arguments are unifiable by σ, and returns undef if unification is not possible. Thus,

$$\text{unify[actuals, \$arglist}_i\text{]} = \sigma_i$$
$$\blacksquare$$
$$\text{match[actuals, newversion[\$arglist}_i\text{]]} = \sigma_i \wedge \sigma_i \neq \text{undef}$$

QED

Lemma 3

$$S[\$subgoals] \equiv try[\$subgoals] = \sigma \wedge \sigma \neq undef$$

Proof:

Each subgoal is a function application P(x1,...,xn) that is true if and only if the precondition on the input arguments is true and there exists a substitution σ such that $(x1,...xn)\sigma$ is the unique output tuple associated with (x1,...,xn) by P. For the conjunction of a list of subgoals to be true, each must be true, and, since all their variables are bound by the same universal quantifier, a value supplied to a variable in the evaluation of one subgoal is propagated to all occurrences of that variable throughout the list of subgoals. Thus, S[*$subgoals*] is true if and only if there exists a substitution σ such that $subgoal_1\sigma \wedge subgoal_2\sigma \wedge ... \wedge subgoal_n\sigma$.

All of this is implicit in the semantics of the intermediate form, simply a property of bound variables. The LISP form makes it all explicit by requiring that evaluation of P(x1,...,xn) = σ where σ is the substitution such that $(x1,...,xn)\sigma$ is the unique tuple associated with (x1,...,xn) by P. If there does not exist any such substitution then try returns undef.

$try[(\$subgoal_1, ..., \$subgoal_n)] =$
  IF $eval[\$subgoal_1] = \sigma \wedge \sigma \neq undef$
    $\wedge\ try[mk\text{-}subst[(\$subgoal_2, ..., \$subgoal_n), \sigma]] = \sigma'$
      $\wedge\ \sigma' \neq undef$
  THEN $\sigma \circ \sigma'$
  ELSE undef

Note that since substitutions are made as you go, and evaluations that cause binding to variables always create new variables to bind to, we know that $\forall x$ such that σ contains a binding, say x/a, there does not exist in σ

any binding of the form x/y where y≠a, or any binding of the form y/z where z contains x. Thus,

$$S[\$subgoals] \equiv try[\$subgoals] = \sigma \wedge \sigma \neq undef.$$

QED

Sublemma 4.1

A recursive definition of the form:

```
f[u] = IF null[u] THEN undef
       ELSE IF P[first[u]] THEN g[first[u]]
            ELSE f[rest[u]]
```

has the property that, if $u = (u_1\ u_2 \ldots u_n)$, then

$$f[u] = a \wedge a \neq undef \equiv \exists i\,(g[u_i] = a \wedge a \neq undef \wedge P[u_i] \wedge (\bigwedge_{j=1}^{i-1} \neg P[u_j]))$$

This is a direct consequence of the definition of f, which specifies that each element of u is tested in the order given, and that no element is tested unless all before it in the list have been tried and have failed. We prove it by induction on the length of the list u.

Basis: $u = (u_1)$

```
f[(u1)] = IF null[(u1)] THEN undef
                ELSE IF P[u1] THEN g[u1]
                     ELSE f[()]
        = IF false THEN undef
                ELSE IF P[u1] THEN g[u1]
                     ELSE IF null[()] THEN undef
                          ELSE ...
        = IF P[u1] THEN g[u1]
                ELSE undef
```

Thus, $f[(u_1)] = a \wedge a \neq undef \equiv P[u_1] \wedge g[u_1] = a \wedge a \neq undef$

Induction Step: Assume that for all lists of length ≤ n:

$$f[u] = a \wedge a \neq undef \equiv \exists i\,(g[u_i] = a \wedge a \neq undef \wedge P[u_i] \wedge (\bigwedge_{j=1}^{i-1} \neg P[u_j]))$$

$$f[(u_1 \ldots u_{(n+1)})] =$$

```
IF null[(u1 ...u(n+1))] THEN undef
```

ELSE IF $P[u_1]$ THEN $g[u_1]$
ELSE $f[(u_2 \ldots u_{(n+1)})]$

$\blacksquare$

IF false THEN undef
ELSE IF $P[u_1]$ THEN $g[u_1]$
ELSE $f[(u_2 \ldots u_{(n+1)})]$

$\blacksquare$

IF $P[u_1]$ THEN $g[u_1]$
ELSE $f[(u_2 \ldots u_{(n+1)})]$

thus, $f[(u_1 \ldots u_{(n+1)})] = a \wedge a \neq$ undef

$\blacksquare$

$(P[u_1] \wedge g[u_1] = a \wedge a \neq \text{undef})$
$\vee (\neg P[u_1] \wedge f[(u_2 \ldots u_{(n+1)})] = a \wedge a \neq \text{undef})$

$\blacksquare$

$(i=1 \wedge g[u_i] = a \wedge a \neq \text{undef} \wedge P[u_i] \wedge (\wedge_{j=1}^{i-1} a_{[j=1]}\ \neg P[u_j]))$
$\vee\ (\neg P[u_1] \wedge \exists i(\ g[u_i] = a \wedge a \neq \text{undef} \wedge P[u_i] \wedge (\wedge_{j=1}^{i-1} a_{[j=2]}\ \neg P[u_j])))$

$\blacksquare$

$\exists i(\ g[u_i] = a \wedge a \neq \text{undef} \wedge P[u_i] \wedge (\wedge_{j=1}^{i-1} a_{[j=1]}\ \neg P[u_j]))$

QED

Lemma 4

$\exists i$ (unify[\$id-list, \$arglist$_i$] = σ_i
$\wedge\ \wedge_{j=1}^{i-1}$ (unify[\$id-list, \$arglist$_j$] = $\sigma_j \rightarrow \neg S[\$subgoals_j\sigma_j]$)
$\wedge\ \ S[\$subgoals_i\sigma_i]$)

$\blacksquare$

bktrkcond[actuals, ((\$arglist$_1$ (cons 'try \$subgoals$_1$))
(\$arglist$_2$ (cons 'try \$subgoals$_2$))
...
(\$arglist$_n$ (cons 'try \$subgoals$_n$)))] = σ
$\wedge\ \sigma \neq$ undef

proof:

Let \$alternatives = ((\$arglist$_1$ (cons 'try \$subgoals$_1$))
(\$arglist$_2$ (cons 'try \$subgoals$_2$))
...
(\$arglist$_n$ (cons 'try \$subgoals$_n$)))

then, bktrkcond[actuals, \$alternatives] =
IF null[\$alternatives] THEN undef
ELSE IF match[actuals, newversion[\$arglist$_1$]] = $\sigma_1 \wedge \sigma_1 \neq$ undef
$\wedge$ try[\$subgoalist$_1\sigma_1$] = $\sigma_1' \wedge \sigma_1' \neq$ undef
THEN cleanup[$\sigma_1 \circ \sigma_1'$, actuals]
ELSE bktrkcond[actuals, ((\$arglist$_2$ cons['try \$subgoals$_2$])
...
(\$arglist$_n$ cons['try \$subgoals$_n$]))]

thus, by Sublemma 4.1

bktrkcond[actuals, (($arglist$_1$ (cons 'try $subgoals$_1$))
($arglist$_2$ (cons 'try $subgoals$_2$))
...
($arglist$_n$ (cons 'try $subgoals$_n$)))] = σ
$\wedge\ \sigma \neq$ undef

■

$\exists i$ match[actuals, newversion[$arglist$_i$]] = $\sigma_i \wedge \sigma_i \neq$ undef
$\wedge$ try[$subgoalist$_iσ_i] = $\sigma_i' \wedge \sigma_i' \neq$ undef
$\wedge\ \sigma$ = cleanup[$\sigma_i \circ \sigma_i'$, actuals] $\wedge\ \sigma \neq$ undef
$\bigwedge_{j=1}^{i-1}$ $\neg$(match[actuals,newversion[$arglist$_j$]] = σ_j
$\wedge\ \sigma_j \neq$ undef $\wedge$ (try[$subgoalist$_jσ_j] = σ_j'
$\wedge\ \sigma_j' \neq$ undef))

■

$\exists i$ match[actuals, newversion[$arglist$_i$]] = $\sigma_i \wedge \sigma_i \neq$ undef
$\wedge$ try[$subgoalist$_iσ_i] = $\sigma_i' \wedge \sigma_i' \neq$ undef
$\wedge\ \sigma$ = cleanup[$\sigma_i \circ \sigma_i'$, actuals] $\wedge\ \sigma \neq$ undef
$\bigwedge_{j=1}^{i-1}$ ((match[actuals,newversion[$arglist$_j$]] = σ_j
$\wedge\ \sigma_j \neq$ undef) $\rightarrow$ $\neg$(try[$subgoalist$_jσ_j] = σ_j'
$\wedge\ \sigma_j' \neq$ undef))

by Lemma 2 we know

unify[actuals, $arglist$_i$] = σ_i
■ match[actuals, newversion[$arglist$_i$]] = $\sigma_i \wedge \sigma_i \neq$ undef

by Lemma 3, we have
S[*$subgoals*] ■ try[$subgoals] = $\sigma \wedge \sigma \neq$ undef

thus, we have

$\exists i$ (unify[$id-list, $arglist$_i$] = σ_i
$\wedge\ \bigwedge_{j=1}^{i-1}$ (unify[$id-list, $arglist$_j$] = $\sigma_j \rightarrow \neg$S[*$subgoals*$_j\sigma_j$])
$\wedge\ \bigwedge$($subgoalist$_i$)$\sigma_i$)

■

bktrkcond[actuals, (($arglist$_1$ (cons 'try $subgoals$_1$))
($arglist$_2$ (cons 'try $subgoals$_2$))
...
($arglist$_n$ (cons 'try $subgoals$_n$)))] = σ
$\wedge\ \sigma \neq$ undef

QED

THEOREM (Correctness of LISP form)

$\forall \bar{x}$ defined[inputs[$id-list, $input-pattern]] $\wedge$ $precondition $\rightarrow$

$$
\begin{aligned}
&[\ \exists i\ (\ \text{unify}[\$\text{id-list}, \$\text{arglist}_i] = \sigma_i \\
&\quad \wedge\ \textstyle\bigwedge_{j=1}^{i-1} (\text{unify}[\$\text{id-list}, \$\text{arglist}_j] = \sigma_j \rightarrow \neg S[\$subgoals_j\sigma_j]) \\
&\quad \wedge\ S[\$subgoals_i\sigma_i]\ ) \\
&\qquad \rightarrow\ (\$\text{id-list})\sigma_i \in \$\text{name})\ ] \\
&\wedge \\
&\forall \bar{x}\ [(\$\text{id-list} \in \$\text{name}) \rightarrow \$\text{postcondition}]
\end{aligned}
$$

■

$$
\begin{aligned}
&\forall \bar{x}\ [\ \text{true-precond}[(\text{cons } \$\text{name actuals})] \\
&\qquad \wedge\ \text{bktrkcond}[\text{actuals}, ((\$\text{arglist}_1\ (\text{cons 'try } \$\text{subgoals}_1)) \\
&\qquad\qquad (\$\text{arglist}_2\ (\text{cons 'try } \$\text{subgoals}_2)) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (\$\text{arglist}_n\ (\text{cons 'try } \$\text{subgoals}_n)))] = \sigma \\
&\qquad \wedge\ \sigma \neq \text{undef} \\
&\qquad\qquad \rightarrow (\text{actuals})\sigma \in \$\text{name}\ ] \\
&\wedge\ \forall \bar{x}\ [(\text{actuals} \in \$\text{name}) \rightarrow \$\text{postcondition}]
\end{aligned}
$$

Proof:

The proof has actually all been accomplished through the lemmas. We take it as obvious that the second conjunct of the first expression is equivalent to the second conjunct of the second expression. Lemmas 1 and 4 supply the necessary parts to put together the equivalence of the first conjuncts of each expression.

Adding a New Target Language

In this section we describe how to add a new target language to the system. The first and most important step is to choose an implementation strategy.

One has a great deal of flexibility at this point. A balance must be found between compatibility with the full power of the specification language and with programs written directly in the target language. The main objective is to generate correct programs in the target language - not to extend or re-define the capabilities of that language. We would like the programs written from their specifications to be able to interact in well-defined and convenient ways with programs written directly in the target language.

If the target language distinguishes functions from procedures, then it may be preferable to use procedures as the implementation of all functions. In this way the

variable bindings are simply a side effect of the computation, which is the way it happens in logic anyway. If one wanted to stay close to logic, all functions would be implemented as boolean functions. Once a strategy has been decided upon the following functions/procedures should be implemented. All of these definitions are to reside in the target language system except for *make_(language name)_def* and *autopred*. These two functions are meant to be added to the program generation system. The only change to be made in the system is in the function *translate*. For type-free languages, the clause *((eq target 'language) (make_language_def))* is added; for typed languages, the clause *((eq target 'language) (mk-strong-typed) (make_language_def))* is added.

1. Primitives:

The functions and predicates listed here are labeled "primitives" because we assume they need not be further defined. The set of primitives given here is neither minimal nor exhaustive, just convenient. One's target language may supply more than these, or less.

If one supplies more primitive types for a strong-typed language, then the system must be informed of those type names. This may be accomplished by type specifications with empty bodies, i.e. when asked for the body part of the specification, simply type a period. Note that this is only for type definitions, and only for strongly-typed languages. For type-free target-languages, and for all other function definitions, if one wishes to make use of a function or predicate already defined in the target language, then one should use the "%*f*" or "%*p*" facility (see page 82).

If a target language is unable to supply some of the following primitives, one may still use the system, but should not make use of those undefined primitives in any specifications.

a. Predicates - primitive predicates do not have an output variable. They are defined in the target language such that they always take true or false as value, in type-free languages, and, if the language is typed then they must either be primitive or defined types in that language. Occurrences of these type predicates in preconditions and postconditions will simply become declarations in a typed target-language.

Integer(x)
Real(x)
Boolean(x)
Is-String(x)
Is-List(x)
binary relational predicates: =, ≠, >, <, ≥, ≤
(I/O): Firstsym(x y z) - meaning y is the first symbol, i.e.token, of x, leaving z. [14]
Firstexp(x y z) - meaning y is the first expression on x, leaving z, where
expression ::= *constant* | *variable* |"(" *expression** ")"
Write(x) - has the value true for all x, and has the side effect of printing the value of x on the current output device.

b. Functions - these are functions whose application to arguments result in terms. Thus, they do not carry output variables. The predicate-ized version of any of these (see page 82) is obtained by preceding the name with "%f"

(Integer): +, -, *, /, rem
(Real): r+, r-, r*, r/
(Boolean): ∧, ∨, ¬
(String): string(l)-makes a string out of a list of characters
s-cat(s1 s2) - string concatenation
s-cons(c s) - adds a character to the front of a string

[14]The specification of the system is done in such a way that *Firstsym(x, y, z)* could simply be implemented as a scanner which gets the first token of input, binds it to *y*, and ignores *x* and *z*. *Firstsym* with three arguments indicates unlimited backtracking abilities over input. Since this is not always implementable in one's target language, one must use the general form judiciously. Similarly with *Firstexp*.

firstch(s) - gets the first character of a string
tail(s) - rest of a string (without first character)
mk-string(c) - makes a string of a single character
(List): first(l) - gets first element of a list l
rest(l) - gets the list l without the first element
cons(x l) - adds x to front of list l
list(x1 ... xn) - creates a list with elements x1 ... xn

c. Constants - the grammar for the intermediate form describes the constants of the system, and of course any mapping to another language must be able to recognize constants. The reason we mention them here specifically, is that we have included at least one constant that is not universally available in typical programming languages. This constant is of course *undef*. It may not be convenient (or even possible) to introduce such a constant in strongly typed languages. However, the purpose of *undef*, i.e., some way of indicating that we have determined that a well-defined value in the appropriate domain does not exist, should be implemented. More is said about this in the section on implementing strongly-typed languages.

2. The following functions must also be implemented. The first, *make_(language name)_def*, should be an actual function or procedure name. The others are indicative of what needs to be accomplished. They need not exist as explicitly defined functions. For example, the target language does not need to have a procedure named *bktrkcond*, however, the function of *bktrkcond*, i.e., the selection of alternatives to attempt in order to complete a computation, must be accomplished by some means.

a. *make_(language name)_def* - This function has available all the information of the intermediate language definition. It creates the target language definition, i.e., the actual syntax of a procedure declaration/definition in the target language.

b. *bktrkcond* - This function may be accomplished as an ordinary conditional, allowing several options, or as a sequence of IF-THEN-ELSE's with added conditions to ensure completion of only one alternative.

1) *undef* - A distinguishable bottom element of every type is used as an undefined element. Languages that are strong-typed without the ability to add a single element to an existing type cause trouble here. In these cases, a variable *undef* may be used which is local to each *bktrkcond* and passed through to *try*.

c. *match* - The pattern match may simply be a unification algorithm; a more general matcher will allow more complicated input to the matcher, and thus enhance the system's efficiency if done well. A simple unification algorithm is guaranteed to be correct, but requires that one must only desire syntactic matching. That is, the matcher would not be able handle the unification of any data types that have more than one representation.

d. *try* - The function *try* attempts to complete the computation of a list of procedure calls. If any call is unsuccessful, then *try* fails, and another alternative is selected by *bktrkcond*.

e. a precondition checker - This may be implemented as a type check and/or first condition of the procedure body, governing the execution of the rest of the body.

Each of the functions mentioned above may require several subfunctions. For example, in a typed language one would probably split *make_(language name)_def* into sub-parts such as *make_dec_part* and *make_body_part*. One also needs to check whether the lexicon of the intermediate language is compatible with the lexicon of the target language, and provide a mapping from one to the other if they are not. Of particular concern here is the fact that the *autopred* facility allows identifiers whose first characters are the symbol ? and the internalized form of the formal parameters are identifiers with first symbol "!".

3. Define *autopred* to allow use of language-system-defined functions and procedures (see page 82). Also helps in the automatic predicatizing of the primitive "functions" mentioned in step 1.

10.1 Implementation Strategy for LISP

The definitions of the primitives for LISP are given in Section 16.1, beginning on page 156. The choice of an implementation strategy was changed after seeing the results in the first implementation of LISP. We discuss this here since it the lesson learned in the process may be applicable to several languages.

Every function call in most implementations of LISP returns a single value. [15] Thus, in the original implementation it was thought that to be consistent with LISP every function call should return the value of its output variable. This decision seemed quite natural at first, in fact it seemed the only way to be compatible with ordinary LISP.

No problems arose as long as the functions being defined had precisely one output variable. It somehow seemed reasonable to expect that one should only want to compute a single value when dealing with a language in which that is the norm. However, when faced with a function having two or more output variables, one had to choose which was to be the value returned. The arbitrary choice was made that the first output variable would carry the value returned by the function call. Although somewhat dissatisfying this choice caused no major problems, immediately.

The point at which the strategy was found to be inadequate came when implementing the propagation of the values of the output variables to the rest of the current list of subgoals. It was certainly possible to do this, but it was not pretty.

A new strategy was chosen. Each function call would return a substitution. This strategy is completely compatible with LISP in that a substitution is indeed a single value. It was still easy to automatically generate "predicate-ized" versions of LISP-system functions.

The new implementation turned out to be much cleaner and more amenable to

---

[15] See *Anatomy of LISP* [A 78] for a discussion of an implementation of LISP with multiple-valued functions.

proof than the old. The lesson to be learned from this is that there are usually many possible choices of implementation strategy, one should not feel stuck with the first choice, and one should not overly restrict the implementation by avoiding what may at face value appear to be an extension of the target language, but in fact is not.

10.2 Implementation of Pascal

In any typed language, some additional information about the types of parameters and local variables is desired, and it is convenient to generate this information once and keep it where one can continually reference it. When it is recognized that a program is to be generated in a strongly typed language we add more information to the property list of the function being defined.

A list of the formal parameters and their types is put under the property *types*. These types are gleaned from the precondition and postcondition specifications. The remainders of the precondition and postcondition (i.e. that which is in addition to type specifications) are listed on the *typedprecond* and *typedpostcond* properties of the function.

The body of the function is searched to determine the names of all functions that are called by the function being defined, and this list of names is stored under the *external-procs* property. A list of the local variables and their types is generated and stored under the *local-decs* property. When an external declaration is made for a

function, it is also kept so that the same declaration need not be derived again for every other function that calls it.

In Pascal, a function or type specification is implemented as a boolean function with value parameters corresponding to the input parameters and *var* parameters corresponding to the output parameters. A function will return the value *true* if it is successful and *false* if it is not. All user-defined types are treated as one type, and functions are generated to distinguish among them.

If one wishes to translate to Pascal, or any strongly typed language, part of the precondition must be a type specification for each input parameter, and part of the postcondition must be a type specification for each output parameter. This means that there may be some specifications that are legal input to the system that will not be translatable into Pascal.

The necessary additions to the system for implementing Pascal as a target language may be found in Section 16.2, page 172. Appendix A, which contains several sample specifications and their translations into LISP programs, exhibits only one generated Pascal program. This is due to the extreme ugliness of the implementation of the "back end" for Pascal. To satisfy the type restrictions of Pascal we had to either disallow user-defined types or map all types to a general type (we used "term"). Mapping all types to one results in a very general structure. This structure is nice theoretically in its universality, but unwieldy in practice. We felt the point could be shown by exhibiting the translation of the factorial function. The full generality of the structure is used even for this simple example, and we are unable to

simplify it for the cases in which we are only dealing with the base types of Pascal. The primitive functions and predicates are defined to take advantage of the operations possible on base types; however, this is because the primitives are implemented by hand and can be designed individually.

In conclusion, the implementation of Pascal has shown that strongly-typed languages may be added to the system; however, if the system is to be considered for "practical" use, an implementation that encodes types more efficiently should be investigated.

Implementation Notes

The top level of the system is called "top" and is called by typing "(top)" to MACLSP. When the program is initialized it prompts the user suggesting that if help is needed one should type "?". This results in instructions for input specifications. The system prompts the user for individual parts of a specification, and again a "?" response will provide information about the required specification, along with an example.

When the user terminates a session, by typing "." at the top level, the system asks if one wishes to save the definitions of the session on a file, and asks for a filename if the answer is affirmative. Before terminating the session, the system informs the user how to include the definitions just saved the next time the system is started.

The pattern-matcher plays an important part in the computation of programs.

Alternatives in the backtracking-conditional are chosen by matching the actual parameter list against a pattern. A straight unification algorithm is easy to implement but not always as powerful as we would like. It works well where there is a unique representation of the objects we are trying to match. If one chooses to define a data type in which the representation for each element is not unique, then one should also define an equality predicate "*Equal-<type>*" for the type, and if one desires more than syntactic matching to occur, a function "*Equal-bind-<type>*" that will attempt to match two elements of the type. All other equality testing and matching is done syntactically.

For example, if one chose to represent sets as unordered constructions rather than imposing some order, one might provide the following specifications:

```
type Set
body? Set(Mt-set, true)
  Set( Add-elem(y,X), true) ← Set(X, true), Member(y, X, false).

function Equal-set
input-pattern? (1 1 0)
parameter-list? (x y z)
precondition? Set(x, true) ∧ Set(y, true).
postcondition? Boolean(z, true).
body?
Equal-set(x, y, true) ← Subset(x, y, true), Subset(y, x, true).
```

Of course, "*Subset*" and "*Member*" must also be defined.

Note the use of *false* in the second clause of the specification of the body of *Set*. Negation of predicates is not allowed by the syntax of Horn clauses; using a truth-valued output variable we are able to incorporate negative tests into the language. In the definition of *Member* we would have a clause: *Member(x, (), false)* ←

The *true* or *false* used as an argument of a predicate must be considered a constant (or 0-ary function symbol) not a predicate. When using the constant predicate, *TRUE* or *T*, we distinguish it here by capitalization. The user should be aware of the distinction, however, it is not necessary to communicate it to the system through capitalization since the distinction can be determined by context.

Failure to find an answer will cause the value *undef* to be returned. This is an indication that the answer is undefined either because we attempted to apply a predicate to arguments not in its domain (as specified by the precondition) or we failed to match on all of the clauses of the definition. Returning *undef* will fail the subgoal it is returned to and thus fail the current clause and the next alternative will be attempted. At this level, *undef* is denoting failure to successfully terminate a computation.

At another level, *undef* is a constant that is assumed to be in every domain. It is possible that one might call a subgoal with the constant *undef* in an output variable position; in this case, if *undef* is the value that would be bound to that position, then the subgoal will succeed. There are times when *undef* is the appropriate answer to be returned from a function. For example, if one defines a look-up function that takes a name and a list of name-value pairs, and returns the value associated with the name, then one would expect an output value *undef* if the name does not occur in the list.

The system will provide default values for the parameter-list, precondition, and postcondition of the specification of a function as it does for a type specification. This

makes the input of specifications easier but is not recommended for general use. Even when specifying a program that is only expected to be partially correct, one should be able to provide a precondition that would at least keep some bad inputs from being accepted.

A further convenience for the user, the system will automatically "predicate-ize" functions that are pre-defined in the target-language system. The user indicates such functions to the synthesis system by prefixing the name of a function with "*%f*" and the name of a system predicate (boolean valued function) with "*%p*". An output variable is added to the argument list and the new predicate may be used as any other. For example, the recursive clause for the definition of factorial:

$$fact(x, y) \leftarrow \%fsubl(x, x1), fact(x1, y1), \%f*(x, y1, y)$$

makes use of the system functions "*subl*" and "*". Again, since these are being defined automatically, the preconditions are simply "true", so the only type checking done will be that provided by the target language. We do not guarantee correctness for definitions made in terms of system functions.

The mapping to a strongly-typed language requires that more detail be spelled out. When the system recognizes that the target language requires explicit declaration of types, it derives from the specifications some additional properties that will be useful in the translation. These include a list of formal parameters associated with their types, a list of local variables and their types, and a list of all functions that are called by the one being specified.

Conclusions and Further Research

We have shown, through proofs of the correctness of the mappings involved, that the system described in this document provides a valid way of generating correct programs. The system works by adding control information to the logic that is specified by the user. The user is still left with the task of inventing the computational logic description of the program desired. We feel that this is reasonable and an advance in the process of obtaining a correct program since the programmer is no longer burdened with the problem of describing the flow of control of the program.

The system as described is "reasonably" target-language-independent. We qualify this statement only because it is easier to translate to some languages than to others. For ease of translation, the target language should allow recursion. The

"back-end" necessary to translate to a non-recursive language, although certainly feasible, would be more complicated than that for a language allowing recursion due to the necessity of translating recursive algorithms. We feel this is almost no restriction at all since we believe that recursion is important enough to be a minimum requirement for any language to be considered "reasonable".

The type structure of a target language is an important factor in determining the ease of its addition to the system. The simplest languages to deal with are those that are type-free, allowing type specifications to be translated into functions that are recognizers for the type being defined. In a typed language, one often feels the need (or desire) for polymorphic functions as allowed by LCF. LCF checks the consistency of types without insisting on knowing precisely the type of every object at compile time. For example, one can define the function *compose*, which takes two arguments of functional types and returns a value of functional type. The type assigned to *compose* by LCF is:([type2 → type3] X [type1 → type2]) → [type1 → type3]. A simple example of another function that is by nature polymorphic is a symbol table lookup. The type of the result of a lookup should agree with the type of the variable the function is called with. However, there may be several different types of variables and values in the table at any time.

The use of generic functions in the specifications is a new feature making it possible to synthesize several programs from a single specification. Generic functions also provide a convenient tool in defining other functions. The portability of the

system is evidenced by its ability to generate a version of itself.[16] Thus it is easily bootstrapped given the "back-end" for the language desired. The ability to generate itself is also an indication that the system can handle large "practical" problems. Several sample programs have been generated, many are listed in Appendix A.

Several extensions to the system and subjects of further research have suggested themselves along the way. We divide these loosely according to whether they deal with the front-end of the system, the present capabilities of the system, or the back-end required to add a new language.

We would like to add a front-end to the system that would allow more natural input. This includes several extensions of the syntax of the specification language. First, the use of embedded function applications cuts down on the amount of typing necessary. We can trivially make use of function applications that have only a single output variable by a simple syntactic manipulation that replaces the occurrence of these in argument positions by a temporary variable and adds the function application, with that same variable in the output position, to the subgoals of the clause in which it occurs before sending the input to the system. If we are also to

---

[16]The version of the system which was generated from the specification given in Chapter 15 is slow in comparison to the hand-written version. We believe this is largely due to the simple-minded pattern matcher being used, which makes actual substitutions rather than simulating them with binding of variables. This is actually a problem with the implementation of *match* written directly in LISP rather than a problem with the specification.

include functions that have not yet been defined, then we must have a way of distinguishing them from constructor functions [17], which must never be turned into predicates. This is not difficult and can be done in any of a number of ways.

Secondly, we can extend the syntax of Horn clauses to allow mixed conjunctions and disjunctions of positive literals (still no negation allowed) on the right hand side of the "←".

Thirdly, it would be nice to allow full use of Predicate Calculus in the specification of a function. This is a much more difficult problem. Synthesis of programs from general descriptions is being studied by several researchers ([C 77], [CD 78], [CS 77], [D 75], [D 77], [DM 75], [MW 77a], [MW 77c]). Hopefully, their results may be incorporated in this system in the future. Brian Beach, a student at the University of California, Santa Cruz, has implemented an interactive system that helps one derive Horn clauses from more general statements in Predicate Calculus [B 79].

Lastly, we would like to include an interactive program that helps the user derive the specifications for a program and prove the correctness of the specifications. The system could check the completeness of the specification by making sure that at least one Horn clause is applicable to every element of the domain of the function as specified by the precondition. This is difficult in general but for inductively defined domains may simply be a reminder to the programmer that they include basis and

---

[17] Constructor functions are used to define data structures inductively. For example, the definition of type *Set* given on page 21 uses the constructor function *Add-elem*.

constructed elements. To prove the correctness of a specification one must: 1)Prove each Horn clause as a theorem in the problem domain; 2) Prove that the precondition guarantees termination of the program (usually a proof by induction on the input); and 3) Prove that successful termination of the program always results in an answer satisfying the postcondition, and that the answer produced is unique. This is then a proof that we have a correct specification of *a* problem; again, we can never prove it is the problem we had "in mind". It is of course too much to expect that the system be capable of doing all of this on its own, but a semi-automatic verifier or proof checker would be useful.

There are also several extensions to the capabilities of the system that are desirable. We would like to extend the use of generic function specifications to allow selection of individual versions of the function by type of the arguments as well as by input pattern. Another useful feature would be obtained by making arrays a primitive data type. Array access is not achieved efficiently by a logic program; since most available languages offer arrays as a primitive type, we could make use of them in specifications realizing that the fast index algorithms of a target language would eventually be used by the program.

Functional arguments are disallowed in any first-order theory, however, we could circumvent the jump to second order and its associated problems by considering the use of function names as arguments. These names could then indicate where we might find the appropriate definition of the function we wish to use.

Another restriction we have imposed is the functionality of our specifications.

This gave a great savings in terms of the limited backtracking needed for evaluation (over clauses, but never over subgoals). We would like to include a way of indicating that selective subgoal backtracking is desired. This would give us the ability to make statements about the existence of an answer that satisfies several subgoals simultaneously. Each subgoal may be satisfied by a set of answers, we want an element of the intersection of these sets.

There are several ways of improving the efficiency of the generated programs. We intend to incorporate some analysis to determine the best ordering of alternatives and of subgoals within alternatives. We would also like to generate programs to optimize the source language programs for particular target languages. In languages in which recursion is implemented inefficiently, this would include removal of recursion.

The most dramatic improvement to the system would be the development of a program that could automatically generate the mapping from intermediate to target language from a formal specification of the syntax and semantics of the target language. This is similar to the work being done on translator writing systems, the difference being that the target is a high level programming language rather than a machine language.

Bibliography

[A 78] Allen, J. R.,*Anatomy of LISP*. McGraw-Hill Publishing Co., New York, New York, 1978.

[AN 76] Andreka, H. and I. Nemeti, "The Generalised Completeness of Horn Predicate-Logic as a Programming Language", D.A.I. Research Report No. 21, Department of Artificial Intelligence, University of Edinburgh, March, 1976.

[B 57] Backus, J. W., R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt, "The FORTRAN Automatic Coding System", *Proceedings of the Western Joint Computer Conference*, vol. 11, 1957.

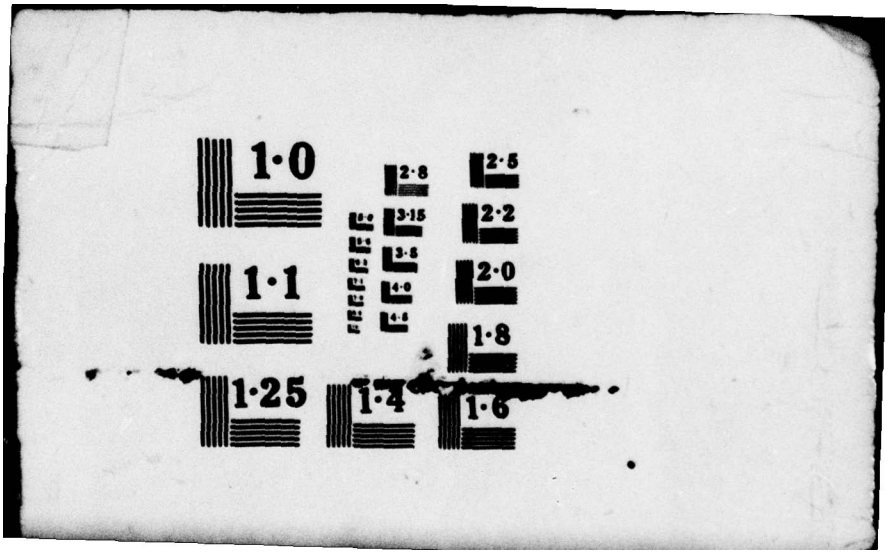
[BGW 77] Balzer, R. M., N. M. Goldman, and D.S. Wile, "Informality in Program Specifications", University of Southern California Information Sciences Institute, ISI/RR-77-59, April, 1977.

[B 77] Bauer, F. L., H. Partsch, P. Pepper, and H. Wössner, "Notes on the Project CIP: Outline of a Transformation System", TUM-INFO-7729, Institut Fur Informatik, Technische Universität München, July, 1977.

[B 79] Beach, B., "Transforming Predicate Calculus Statements Into Horn Clauses", (tentative title), Senior Thesis, University of California, Santa Cruz, June, 1979.

AD-A073 023 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
GENERATING CORRECT PROGRAMS FROM LOGIC SPECIFICATIONS.(U)
MAY 79 R E DAVIS N00014-76-C-0682
UNCLASSIFIED TR-79-5-001 NL

2 OF 3
AD
A073023


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0

[BK 76] Biermann, A. W., and R. Krishnaswamy, "Constructing Programs From Example Computations", *IEEE Transactions on Software Engineering*, vol. 2, no. 3, Sept., 1976.

[B 74] Buchanan, J. R., "A Study in Automatic Programming", AIM-245, STAN-CS-74-458, Ph.D. Thesis, Stanford University, May, 1974.

[BL 74] Buchanan, J. R., and D. C. Luckham, "On Automating the Construction of Programs", SAIL AIM-236, STAN-CS-74-433, Stanford University, May, 1974.

[BD 75] Burstall, R. M., and J. Darlington, "Some Transformations for Developing Recursive Programs", *Proceedings of the International Conference on Reliable Software*, Los Angeles, Ca., 1975.

[BD 76] Burstall, R. M., and J. Darlington, "A Transformation System for Developing Recursive Programs", D.A.I. Research Report No.19, University of Edinburgh, March, 1976.

[C 78] Clark, K., "Negation as Failure", in *Logic and Data Bases* (H. Gallaire and J. Minker, Eds.), Plenum Press, New York, N.Y., 1978.

[C 77] Clark, K., "Synthesis and Verification of Logic Programs", Research report, CCD, Imperial College, 1977.

[CD 78] Clark, K., and J. Darlington, "Algorithm Classification Through Synthesis", Imperial College of Science and Technology, London, June, 1978, to appear in *Computer Journal*.

[CS 77] Clark, K., and S. Sickel, "Predicate Logic: A Calculus for Deriving Programs", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., Aug., 1977.

[D 75] Darlington, J., "Applications of Program Transformation to Program Synthesis", *IRIA, Proceedings of the Symposium on Proving and Improving Programs*, Rocquencourt, France, 1975.

[D 77] Darlington, J., "Program Transformation and Synthesis: Present Capabilities", DAI Report No. 48, University of Edinburgh, Research Report No. 77/43, Imperial College of Science and Technology, Dept. of Computing and Control, Sept., 1977.

[DB 76] Darlington, J., and R. M. Burstall, "A System Which Automatically Improves Programs", *Acta Informatica*, vol. 6, pp.41-60, 1976.

[D 78b] Davis, R. A., "Minis vs. Micros: OEM Decision-Making Risks Grow", *Digital Design*, vol. 8, no. 8, Aug., 1978.

[D 76] Davis, R. E., "Deduction, Truth, and Computation", Master's Thesis, San Jose State University, San Jose, Ca., 1976.

[DM 75] Dershowitz, N., and Z. Manna, "On Automated Structured Programming", *IRIA, Proceedings of the Symposium on Proving and Improving Programs*, Rocquencourt, France, 1975.

[E 78] van Emden, M. H., "Computation and Deductive Information Retrieval", in *Formal Description of Programming Concepts*, (E. Neuhold, Ed.), North Holland, 1978.

[EK 74] van Emden, M. H., and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language", Memorandum MIP-R-103, School of Artificial Intelligence, University of Edinburgh, Feb., 1974.

[F 79] Franusich, M., "Analysis of Logic Programs for Static and Dynamic Subgoal Selection", Senior Thesis, University of California, Santa Cruz, June, 1979.

[G 75] Gerhart, S. L., "Correctness-Preserving Program Transformations", *Proceedings of the 2nd ACM Symposium on Principles of Programming Languages*, pp.54-66, January, 1975.

[GBW 78] Goldman, N. M., R. M. Balzer, and D. S. Wile, "The Inference of Domain Structure from Informal Process Descriptions", *Proceedings of the Workshop on Pattern-Directed Inference Systems*, Honolulu, May, 1978.

[G 77] Green, C. C., "A Summary of the PSI Program Synthesis System", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., August 1977.

[H 75a] Hardy, S., "Synthesis of LISP Functions from Examples", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia, U.S.S.R., Sept., 1975.

[H 76] Heidorn, G. E., "Automatic Programming Through Natural Language Dialogue: A Survey", *IBM Journal of Research and Development*, vol. 20, no. 4, July, 1976.

[H 75b] von Henke, F., "On Generating Programs from Data Types: An Approach to Automatic Programming", *IRIA, Proceedings of the Symposium on Proving and Improving Programs*, Rocquencourt, France, 1975.

[H 79] von Henke, F., "Pascal Unifiers", unpublished manuscript, 1979.

[ILL 75] Igarashi, S., R. L. London, and D. C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation", *Acta Informatica*, vol. 4, pp. 145-182, 1975.

[JW 74] Jensen, K., and N. Wirth, *Pascal User Manual and Report*, second edition, Springer-Verlag, New York, N.Y., 1974.

[K 52] Kleene, S. C.. *Introduction to Metamathematics*. Van Nostrand Company, Inc., Princeton, N.J., 1952.

[K 74] Kowalski, R., "Predicate Logic as Programming Language", *Proceedings IFIP 74*, North-Holland Publishing Company, 1974.

[LC 74] Lee, R. C. T., and S. K. Chang, "Structured Programming and Automatic Program Synthesis", *Proceedings of a Symposium on Very High Level Languages*, Santa Monica, Ca., 1974.

[L 75] Lenat, D. B., "Synthesis of Large Programs from Specific Dialogues", *IRIA, Proceedings of the Symposium on Proving and Improving Programs*, Rocquencourt, France, 1975.

[L 77] Loveman, D. B., "Program Improvement by Source-to-Source Transformation", *Journal of the ACM* 24:1, pp.121-145, 1977.

[L 73] Lukasik, S. J., Testimony to House Armed Services Subcommittee, USGPO, pp.3509-3559, June, 1973.

[MW 77a] Manna, Z. and R. Waldinger, "The Automatic Synthesis of Recursive Programs", *Proceedings of the Symposium on AI and Programming Languages*, SIGPLAN/SIGART NOTICES/NEWSLETTER Vol. 12, No. 8, August, 1977/ No. 64, August, 1977.

[MW 77b] Manna, Z. and R. Waldinger, "The Logic of Computer Programming", AIM-298, STAN-CS-77-611, Stanford University, August, 1977.

[MW 77c] Manna, Z. and R. Waldinger, "Synthesis: Dreams => Programs", AIM-302, STAN-CS-77-630, Stanford University, November, 1977.

[MT 79] McCarthy, J., and C. Talcott, *LISP Programming and Proving*. To be published.

[M 64] Mendelson, E., *Introduction to Mathematical Logic*. D. Van Nostrand Company, Inc., Princeton, New Jersey, 1964.

[M 74] Moon, D., *MACLISP Reference Manual*, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1974.

[S 77a] Sickel, S., "A Logic-Based Programming Methodology", Technical Report no. 77-8-001, Univesity of California, Santa Cruz, March, 1977.

[S 78] Sickel, S., "Invertibility of Logic Programs", Technical Report No. 78-8-005, University of California, Santa Cruz, August, 1978.

[S 65] Slagle, J. R., "Experiments with a Deductive Question-Answering Program", *Communications of the Association for Computing Machinery*, Vol. 8, No. 12, Dec., 1965.

[S 74] Standish, T., "ARPA's Automatic Programming Research: A Quest for a Coherent View", (draft), Feb., 1974.

[S 77b] Summers, P. D., "A Methodology for LISP Program Construction from Examples", *Journal of Association for Computing Machinery*, vol. 24, no. 1, Jan., 1977.

[W 69] Waldinger, R. J., "Constructing Programs Automatically Using Theorem Proving", Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., May, 1969.

[WBG 77] Wile, D. S., R. M. Balzer, and N. M. Goldman, "Automated Derivation of Program Control Structure from Natural Language Program Descriptions", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester, N.Y., Aug., 1977.

[UM 77] Ulrich, J. W., and R. Moll, "Program Synthesis by Analogy", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester, N.Y., Aug., 1977.

Appendix A: Sample Specifications

```
type Nat
body? Nat(x true) ← Integer(x true), ≥(x 0 true).

(DEFUN NAT FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'NAT L))
        (BKTRKCOND L
            '(((IX T) (TRY (INTEGER IX T) (≥ IX 0 T))))))
        (T 'UNDEF)))


generic add
parameter list? (x y z)
choices?
    (1 1 0)
function name? %f+                    since %f+ is a function
choices?                              already known to the system, no
    (1 0 1)                           further information is required
function name? add-2
precondition? Integer(x) ∧ Integer(z).
postcondition? Integer(y).
body-name? sub-2
choices?
```

```
      (0 1 1)
function name? add-1
precondition? Integer(y) ∧ Integer(z).
postcondition? Integer(z).
body-name? sub-1
choices?

body-defs:
sub-2?
add(x y z) ← %f-(z x y).

sub-1?
add(x y z) ← %f-(z y x).

(PUTPROP 'ADD '(((1 1 0) . %F+) ((1 0 1) . ADD-2) ((0 1 1) . ADD-1)) 'GENERIC)

(DEFUN ADD-2 FEXPR (L)
     (COND ((TRUE-PRECOND (CONS 'ADD-2 L))
          (BKTRKCOND L
                '(((IX T) (TRY (INTEGER IX T) (≥ IX 0 T))))))
          (T 'UNDEF)))

(DEFUN ADD-1 FEXPR (L)
     (COND ((TRUE-PRECOND (CONS 'ADD-1 L))
          (BKTRKCOND L
                '(((IX T) (TRY (INTEGER IX T) (≥ IX 0 T))))))
          (T 'UNDEF)))

END_OF_GENERIC_SPEC
```

```
generic imult
parameter list? (x y z)
choices?
      (1 1 0)
function name? %f*
choices?
     (0 1 1)
function name? imult-1
precondition? Integer(y) ∧ Integer(z).
postcondition? Integer(x).
body-name? idiv-1
choices?
      (1 0 1)
function name? imult-2
precondition? Integer(x) ∧ Integer(z).
```

Integer multiplication, the result will be an integer, or *undef if there* is no integer meeting the semantics.

Ordinary multiplication, as above the system already knows it.

```
postcondition? Integer(y).
body-name? idiv-2
choices?

body-defs:
idiv-1?
imult(0 x 0)
imult(undef 0 x) ← =(x 0)
imult(x y z) ← %fl(z y x), imult(x y z).
```

This may look strange but it will be able to sort out which imult to use.

```
idiv-2:
imult(x 0 0)
imult(0 undef x) ← =(x 0)
imult(x y z) ← %fl(z x y), imult(x y z).

(PUTPROP 'IMULT '(((1 1 0) . %F*) ((0 1 1) . IMULT-1) ((1 0 1) . IMULT-2))
         'GENERIC)

(DEFUN IMULT-1 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'IMULT-1 L))
           (BKTRKCOND L
                      '(((IX T) (TRY (INTEGER IX T) (≥ IX 0 T))))))
          (T 'UNDEF)))

(DEFUN IMULT-2 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'IMULT-2 L))
           (BKTRKCOND L
                      '(((IX T) (TRY (INTEGER IX T) (≥ IX 0 T))))))
          (T 'UNDEF)))


END_OF_GENERIC_SPEC
```

the above definitions rely heavily on the arithmetic already defined in the target language. The following definitions may be considered more typical.

```
function gcd                                    gcd(x y z) means that the greatest
input-pattern? (1 1 0)                          common divisor of x and y is z.
parameter list? (x y z)
precondition? Nat(x true) ∧ Nat(y true).
postcondition? Nat(z true).
body? gcd(0 0 undef)
      gcd(0 x x)
      gcd(x 0 x)
      gcd(x y z) ← ≥(x y), add(y w x), gcd(w y z)
      gcd(x y z) ← ≥(y x), add(x w y), gcd(x w z).
(DEFUN GCD FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'GCD L))
        (BKTRKCOND L
                '(((0 0 UNDEF) (TRY))
                  ((0 !X !X) (TRY))
                  ((!X 0 !X) (TRY))
                  ((!X !Y !Z)
                   (TRY (≥ !X !Y)
                        (ADD-2 !Y !W !X)
                        (GCD !W !Y !Z)))
                  ((!X !Y !Z)
                   (TRY (≥ !Y !X)
                        (ADD-2 !X !W !Y)
                        (GCD !X !W !Z))))))
          (T 'UNDEF)))
```

```
generic factor                                  factor(w n p r) means w=n^p*r
parameter list? (w n p r)                       and p is maximal
choices?
      (0 1 1 1)
function name? factor-1
precondition? Nat(n true) ∧ Nat(p true) ∧ Nat(r true).
postcondition? Nat(w true).
body-name? mult-out
choices?
      (1 1 0 0)
function name? factor-34
precondition? Nat(w true) ∧ Nat(n true).
postcondition? Nat(p true) ∧ Nat(r true).
body-name? ss-factor
choices?

body-defs:
mult-out?
factor(r n 0 r) ← gcd(r n 1)
```

```
factor(w n p r) ← imult(n r x), zfsubl(p p1), factor(w n p1 x).

ss-factor?
factor(w n 0 w) ← <(w n)
factor(w n 0 w) ← gcd(w n 1)
factor(w n p r) ← ≥(w n), imult(v n w), factor(v n p1 r), add(p1 1 p).

(PUTPROP 'FACTOR '(((0 1 1 1) . FACTOR-1) ((1 1 0 0) . FACTOR-34))
         'GENERIC)

(DEFUN FACTOR-1 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'FACTOR-1 L))
        (BKTRKCOND L
            '(((0 0 UNDEF) (TRY))
              ((0 !X !X) (TRY))
              ((!X 0 !X) (TRY))
              ((!X !Y !Z)
               (TRY (≥ !X !Y)
                    (ADD-2 !Y !W !X)
                    (GCD !W !Y !Z)))
              ((!X !Y !Z)
               (TRY (≥ !Y !X)
                    (ADD-2 !X !W !Y)
                    (GCD !X !W !Z))))))
        (T 'UNDEF)))

(DEFUN FACTOR-34 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'FACTOR-34 L))
        (BKTRKCOND L
            '(((0 0 UNDEF) (TRY))
              ((0 !X !X) (TRY))
              ((!X 0 !X) (TRY))
              ((!X !Y !Z)
               (TRY (≥ !X !Y)
                    (ADD-2 !Y !W !X)
                    (GCD !W !Y !Z)))
              ((!X !Y !Z)
               (TRY (≥ !Y !X)
                    (ADD-2 !X !W !Y)
                    (GCD !X !W !Z))))))
        (T 'UNDEF)))


END_OF_GENERIC_SPEC
```

```
generic E
parameter list? (x n y z)
choices?
      (1 1 1 0)
function name? E-4
precondition? Nat(x true) ∧ Nat(n true) ∧ Nat(y true).
postcondition? Nat(z true).
body-name? ebod-4
choices?
      (1 1 0 1)
function name? E-3
precondition? Nat(x true) ∧ Nat(n true) ∧ Nat(z true).
postcondition? Nat(y true).
body-name? ebod-3
choices?
      (0 1 1 1)
function name? E-1
precondition? Nat(y true) ∧ Nat(n true) ∧ Nat(z true).
postcondition? Nat(x true).
body-name? ebod-1
choices?
.
body-defs:
ebod-4?
E(0 n y 0) ← =(y 0)
E(x n 0 1) ← =(x 0)
E(1 n y 1)
E(x n y z) ← factor(x n p r), imult(p y q), add(n 1 n1), E(r n1 y s), factor(z n q s).

ebod-3?
E(x n 0 1) ← =(x 0)
E(x n y z) ← factor(x n p r), add(n 1 n1), factor(z n q s), imult(p y q), E(r n1 y s).

ebod-1?
E(0 n y 0) ← =(y 0)
E(1 n y 1)
E(x n y z) ← factor(z n q s), imult(p y q), add(n 1 n1), E(r n1 y s), factor(x n p r).

(PUTPROP 'E '(((1 1 1 0) . E-4) ((1 1 0 1) . E-3) ((0 1 1 1) . E-1)) 'GENERIC)
```

```
(DEFUN E-4 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'E-4 L))
        (BKTRKCOND L
            '(((0 0 UNDEF) (TRY))
              ((0 !X !X) (TRY))
              ((!X 0 !X) (TRY))
              ((!X !Y !Z)
               (TRY (≥ !X !Y)
                   (ADD-2 !Y !W !X)
                   (GCD !W !Y !Z)))
              ((!X !Y !Z)
               (TRY (≥ !Y !X)
                   (ADD-2 !X !W !Y)
                   (GCD !X !W !Z))))))
        (T 'UNDEF)))

(DEFUN E-3 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'E-3 L))
        (BKTRKCOND L
            '(((0 0 UNDEF) (TRY))
              ((0 !X !X) (TRY))
              ((!X 0 !X) (TRY))
              ((!X !Y !Z)
               (TRY (≥ !X !Y)
                   (ADD-2 !Y !W !X)
                   (GCD !W !Y !Z)))
              ((!X !Y !Z)
               (TRY (≥ !Y !X)
                   (ADD-2 !X !W !Y)
                   (GCD !X !W !Z))))))
        (T 'UNDEF)))

(DEFUN E-1 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'E-1 L))
        (BKTRKCOND L
            '(((0 0 UNDEF) (TRY))
              ((0 !X !X) (TRY))
              ((!X 0 !X) (TRY))
              ((!X !Y !Z)
               (TRY (≥ !X !Y)
                   (ADD-2 !Y !W !X)
                   (GCD !W !Y !Z)))
              ((!X !Y !Z)
               (TRY (≥ !Y !X)
                   (ADD-2 !X !W !Y)
                   (GCD !X !W !Z))))))
```

```
        (T 'UNDEF)))

END_OF_GENERIC_SPEC


generic exp                                   exp(x y z) means x^y=z
parameter list? (x y z)
choices?
      (1 1 0)
function name? exp-3
precondition? Nat(x) ∧ Nat(y).
postcondition? Nat(z).
body-name? expo
choices?
      (1 0 1)
function name? exp-2
precondition? Nat(x) ∧ Nat(z).
postcondition? Nat(y).
body-name? expo
choices?
      (0 1 1)
function name? exp-1
precondition? Nat(z) ∧ Nat(y).
postcondition? Nat(x).
body-name? expo
choices?
      .
body-defs:
expo?
exp(x y z) ← E(x 2 y z).

(PUTPROP 'EXP '(((1 1 0) . EXP-3) ((1 0 1) . EXP-2) ((0 1 1) . EXP-1))
         'GENERIC)

(DEFUN EXP-3 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'EXP-3 L))
        (BKTRKCOND L
                '(((0 0 UNDEF) (TRY))
                  ((0 !X !X) (TRY))
                  ((!X 0 !X) (TRY))
                  ((!X !Y !Z)
                   (TRY (≥ !X !Y)
                        (ADD-2 !Y !W !X)
                        (GCD !W !Y !Z)))
                  ((!X !Y !Z)
```

```
                    (TRY (≥ !Y !X)
                        (ADD-2 !X !W !Y)
                        (GCD !X !W !Z))))))
            (T 'UNDEF)))

(DEFUN EXP-2 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'EXP-2 L))
        (BKTRKCOND L
                '(((0 0 UNDEF) (TRY))
                 ((0 !X !X) (TRY))
                 ((!X 0 !X) (TRY))
                 ((!X !Y !Z)
                  (TRY (≥ !X !Y)
                      (ADD-2 !Y !W !X)
                      (GCD !W !Y !Z)))
                 ((!X !Y !Z)
                  (TRY (≥ !Y !X)
                      (ADD-2 !X !W !Y)
                      (GCD !X !W !Z))))))
            (T 'UNDEF)))

(DEFUN EXP-1 FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'EXP-1 L))
        (BKTRKCOND L
                '(((0 0 UNDEF) (TRY))
                 ((0 !X !X) (TRY))
                 ((!X 0 !X) (TRY))
                 ((!X !Y !Z)
                  (TRY (≥ !X !Y)
                      (ADD-2 !Y !W !X)
                      (GCD !W !Y !Z)))
                 ((!X !Y !Z)
                  (TRY (≥ !Y !X)
                      (ADD-2 !X !W !Y)
                      (GCD !X !W !Z))))))
            (T 'UNDEF)))


END_OF_GENERIC_SPEC


type is-set
body?
is-set('mt-set,true)
is-set( add-elem(x,s), true) ← is-set(s, true), set-mem(x, s, false).
```

```
(DEFUN IS-SET FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'IS-SET L))
        (BKTRKCOND L
            '((('MT-SET T) (TRY))
              (((ADD-ELEM IX IS) T)
               (TRY (IS-SET IS T)
                   (SET-MEM IX IS FALSE))))))
        (T 'UNDEF)))
```

```
function set-mem
input pattern? (1 1 0)
parameter list? (x y z)
precondition? is-set(y, true).
postcondition? boolean(z, true).
body?
set-mem(x, 'mt-set, false)
set-mem(x, add-elem(x, s), true)
set-mem(x, add-elem(y, s), true) ← set-mem(x, s, true)
set-mem(x, add-elem(y, s), false) ← same(x, y, z), same(z, false, true),
                                    set-mem(x, s, false).
```

```
(DEFUN SET-MEM FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'SET-MEM L))
        (BKTRKCOND L
            '(((IX 'MT-SET FALSE) (TRY))
              ((IX (ADD-ELEM IX IS) T) (TRY))
              ((IX (ADD-ELEM IY IS) T)
               (TRY (SET-MEM IX IS T)))
              ((IX (ADD-ELEM IY IS) FALSE)
               (TRY (SAME IX IY IZ)
                   (SAME IZ FALSE T)
                   (SET-MEM IX IS FALSE))))))
        (T 'UNDEF)))
```

```
function same
input pattern? (1 1 0)
parameter list?
body?
same(x x true)
same(x y false).
```

```
(DEFUN SAME FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'SAME L))
        (BKTRKCOND L
            '(((IX IX T) (TRY)) ((IX IY FALSE) (TRY))))))
```

```
        (T 'UNDEF)))


function union
input pattern? (1 1 0)
parameter list? (x y z)
precondition? is-set(x, true) ∧ is-set(y,true).
postcondition? is-set(z, true).
body?
union('mt-set, y, y)
union(x, 'mt-set, x)
union(add-elem(x,s), y, add-elem(x,z)) ← set-mem(x, y, false), union(s, y, z)
union(add-elem(x,s), y, z) ← set-mem(x, y, true), union(s, y, z).

(DEFUN UNION FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'UNION L))
        (BKTRKCOND L
            '((('MT-SET !Y !Y) (TRY))
              ((!X 'MT-SET !X) (TRY))
              (((ADD-ELEM !X !S) !Y (ADD-ELEM !X !Z))
               (TRY (SET-MEM !X !Y FALSE)
                    (UNION !S !Y !Z)))
              (((ADD-ELEM !X !S) !Y !Z)
               (TRY (SET-MEM !X !Y T)
                    (UNION !S !Y !Z))))))
        (T 'UNDEF)))


type Intlist
body?
Intlist(( ),true)
Intlist(cons(x, l), true) ← Integer(x, true), Intlist(l, true).

(DEFUN INTLIST FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'INTLIST L))
        (BKTRKCOND L
            '(((NIL T) (TRY))
              (((CONS !X !L) T)
               (TRY (INTEGER !X T) (INTLIST !L T))))))
        (T 'UNDEF)))

function Insertsort
input pattern? (1 0)
parameter list? (x y)
precond? Intlist(x, true).
postcond? Intlist(y, true) ∧ Perm(x, y, true).
```

```
body?
Insertsort((),())
Insertsort(cons(x,l), y) ← Insertsort(l, w), Insert(x, w, y).

(DEFUN INSERTSORT FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'INSERTSORT L))
        (BKTRKCOND L
            '(((NIL NIL) (TRY))
              (((CONS !X !L) !Y)
               (TRY (INSERTSORT !L !W)
                    (INSERT !X !W !Y))))))
        (T 'UNDEF)))


function Insert
input pattern? (1 1 0)
parameter list? (x y z)
precond? Integer(x, true) ∧ Intlist(y, true).
postcond? Intlist(z, true).
body?
Insert(x, (), cons(x, ()))
Insert(x, cons(y,l), cons(x, cons(y,l))) ← ?p≤(x, y, true)
Insert(x, cons(y,l), cons(y,z)) ← ?p>(x, y, true), Insert(x, l, z).

(DEFUN INSERT FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'INSERT L))
        (BKTRKCOND L
            '(((!X NIL (CONS !X NIL)) (TRY))
              ((!X (CONS !Y !L) (CONS !X (CONS !Y !L)))
               (TRY (?P≤ !X !Y T)))
              ((!X (CONS !Y !L) (CONS !Y !Z))
               (TRY (?P> !X !Y T) (INSERT !X !L !Z))))))
        (T 'UNDEF)))

function Selectionsort
input pattern? (1 0)
parameter list? (x y)
precond? Intlist(x, true).
postcond? Intlist(y, true) ∧ Perm(x, y, true).
body?
Selectionsort((), ())
Selectionsort(cons(x,u1), cons(y,u)) ← Partition-by-min(cons(x,u1), y, u2),
                                      Selectionsort(u2, u).

(DEFUN SELECTIONSORT FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'SELECTIONSORT L))
```

```
        (BKTRKCOND L
                '(((NIL NIL) (TRY))
                  (((CONS !X !U1) (CONS !Y !U))
                   (TRY (PARTITION-BY-MIN (CONS !X !U1)
                                          !Y
                                          !U2)
                        (SELECTIONSORT !U2 !U))))))
        (T 'UNDEF)))
```

```
function Partition-by-min
input pattern? (1 0 0)
parameter list? (u1 x u2)
precond? Intlist(u1, true) ∧ Non-empty(u1, true).
postcond? Integer(x, true) ∧ Intlist(u2, true).
body?
Partition-by-min(cons(x,()), x, ())
Partition-by-min(cons(x, cons(y,u)), z, cons(y,u1)) ←
      ?p≤(x, y, true), Partition-by-min(cons(x,u), z, u1)
Partition-by-min(cons(x, cons(y,u)), z, cons(x,u1)) ←
      ?p>(x, y, true), Partition-by-min(cons(y,u), z, u1).
```

```
(DEFUN PARTITION-BY-MIN FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'PARTITION-BY-MIN L))
        (BKTRKCOND L
                '((((CONS !X NIL) !X NIL) (TRY))
                  (((CONS !X (CONS !Y !U)) !Z (CONS !Y !U1))
                   (TRY (?P≤ !X !Y T)
                        (PARTITION-BY-MIN (CONS !X !U)
                                          !Z
                                          !U1)))
                  (((CONS !X (CONS !Y !U)) !Z (CONS !X !U1))
                   (TRY (?P> !X !Y T)
                        (PARTITION-BY-MIN (CONS !Y !U)
                                          !Z
                                          !U1))))))
        (T 'UNDEF)))
```

```
function Non-empty
input pattern? (1 0)
parameter list? (l x)
precond? List(l, true).
postcond? Boolean(x, true).
```

```
body?
Non-empty(( ),false)
Non-empty(cons(y, u), true).

(DEFUN NON-EMPTY FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'NON-EMPTY L))
        (BKTRKCOND L
            '(((NIL FALSE) (TRY))
              (((CONS !Y !U) T) (TRY)))))
        (T 'UNDEF)))



function Perm
input pattern? (1 0)
parameter list? (x y z)
precond? List(x, true) ∧ List(y, true).
postcond? Boolean(z, true).
body?
Perm( ( ), ( ), true)
Perm(cons(x,u1), cons(x,u2), true) ← Perm(u1, u2, true)
Perm(cons(x,u1), cons(y,u2), true) ← Delete(x, u2, u3),
                                    Delete(y, u2, u4), Perm(u3, u4, true).

(DEFUN PERM FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'PERM L))
        (BKTRKCOND L
            '(((NIL NIL T) (TRY))
              (((CONS !X !U1) (CONS !X !U2) T)
               (TRY (PERM !U1 !U2 T)))
              (((CONS !X !U1) (CONS !Y !U2) T)
               (TRY (DELETE !X !U2 !U3)
                   (DELETE !Y !U2 !U4)
                   (PERM !U3 !U4 T))))))
        (T 'UNDEF)))


function Delete
input pattern? (1 1 0)
parameter list? (x u1 u2)
precond? List(u1, true).
postcond? List(u2, true) ∧ Perm(cons(x, u2), u1, true).
body?
Delete(x, ( ), undef)
Delete(x, cons(x,l), l)
Delete(x, cons(y,l), undef) ← Delete(x, l, undef)
```

```
Delete(x, cons(y,u1), cons(y,u2)) ← Delete(x, u1, u2).

(DEFUN DELETE FEXPR (L)
    (COND ((TRUE-PRECOND (CONS 'DELETE L))
        (BKTRKCOND L
            '(((!X NIL UNDEF) (TRY))
              ((!X (CONS !X !L) !L) (TRY))
              ((!X (CONS !Y !L) UNDEF)
               (TRY (DELETE !X !L UNDEF)))
              ((!X (CONS !Y !U1) (CONS !Y !U2))
               (TRY (DELETE !X !U1 !U2))))))
        (T 'UNDEF)))


function Fact
input pattern? (1 0)
parameter list? (x y)
precond? Integer(x, true) ∧ Greaterequal(x, 0, true).
postcond? Integer(y, true) ∧ Lessthan(0, y, true).
body? Fact(0 1)
      Fact(x, y) ← Subl(x, x1), Fact(x1, y1), Times(x, y1, y).
```

The above specification for the factorial function results in the following Pascal program.

```
PROGRAM G0002,FACT;
TYPE
    ALLTYPS = (INTEGERTYP, REALTYP, BOOLEANTYP,
               CHARTYP, SYMBOLTYP);

    TERMTYPS = (VARIABLE, CONSTANTTYP, FUNAPP);

    TERM = ↑T1;

    TERMLIST = ↑TL1;

    CONSTANT = ↑C1;

    SYMBOL = ↑SYM1;

    T1 = RECORD
        CASE TTYP:TERMTYPS OF
            VARIABLE: (VR: INTEGER);
            CONSTANTTYP: (CNST: CONSTANT);
            FUNAPP: (FNAME: SYMBOL;
```

```
                    ARGS: TERMLIST)
        END;

    TL1 = RECORD
        NOTEMPTY: BOOLEAN;
        FIRST: TERM;
        REST: TERMLIST
        END;

    C1 = RECORD
        CASE CTYP:ALLTYPS OF
            INTEGERTYP: (IVAL: INTEGER);
            REALTYP: (RVAL: REAL);
            BOOLEANTYP: (BVAL: BOOLEAN);
            CHARTYP: (CVAL: CHAR);
            SYMBOLTYP: (SVAL: SYMBOL)
        END;

    SYM1 = RECORD
        NOTEMPTY: BOOLEAN;
        FIRSTCH: CHAR;
        TAIL: SYMBOL;
        END;

    VARPAIRS = ↑VP;

    VP = RECORD
        NOTEMPTY: BOOLEAN;
        OLD: INTEGER;
        NEW: INTEGER;
        REST: VARPAIRS
        END;

FUNCTION GREATEREQUAL(X, Y: TERM; VAR Z: TERM): BOOLEAN;
    EXTERN;
FUNCTION SUB1(X:TERM; VAR Y: TERM): BOOLEAN;
    EXTERN;
FUNCTION TIMES(X, Y: TERM; VAR Z: TERM): BOOLEAN;
    EXTERN;
FUNCTION OCCUR(X, Y: TERM): BOOLEAN;
    EXTERN;
FUNCTION GENVAR: INTEGER;
    EXTERN;
PROCEDURE REPLACE(X, T: TERM; VAR TML: TERMLIST);
    EXTERN;
PROCEDURE SUBST(X, T: TERM; VAR T1, T2: TERMLIST);
```

```
      EXTERN;
FUNCTION EQSYM(X, Y: SYMBOL): BOOLEAN;
      EXTERN;
FUNCTION EQCONST(X, Y: CONSTANT): BOOLEAN;
      EXTERN;
FUNCTION COPYSYM(OLDSYM: SYMBOL): SYMBOL;
      EXTERN;
FUNCTION COPYTERM(OLDTM: TERM): TERM;
      EXTERN;
FUNCTION COPYTERMLIST(TML: TERMLIST): TERMLIST;
      EXTERN;
FUNCTION COPYCONST(OLDCONST: CONSTANT): CONSTANT;
      EXTERN;
FUNCTION UNIFY(VAR X, Y, ALLX, ALLY: TERMLIST;
            FAILED: BOOLEAN): BOOLEAN;
      EXTERN;
      PROCEDURE LOOKUP(TM: TERM; TBL: VARPAIRS;
                        FOUND: BOOLEAN);
      EXTERN;
      PROCEDURE STANDAPART(TML: TERMLIST;
                        VAR DONETBL: VARPAIRS);
      EXTERN;
FUNCTION FACT(X : TERM ; VAR Y : TERM): BOOLEAN;
VAR
      G0014, G0011, G0010, G0009, G0008, G0007, ACTUALS, COPYACTUALS
, MATCHLIST: TERMLIST;
      G0015, G0012, Z1, W1, W, Z, G0005, G0003: TERM;
      G0016, G0013, G0006, G0004: CONSTANT;
      DONETBL: VARPAIRS;
      FLAG, FAILED: BOOLEAN;

BEGIN
 IF
  (GREATEREQUAL (X, G0003, G0005))
 THEN BEGIN
      NEW(ACTUALS);
      ACTUALS↑.NOTEMPTY := FALSE;
      NEW(G0008);
      G0008↑.NOTEMPTY := TRUE;
      G0008↑.FIRST := Y;
      G0008↑.REST := ACTUALS;
      ACTUALS := G0008;
      NEW(G0007);
      G0007↑.NOTEMPTY := TRUE;
      G0007↑.FIRST := X;
      G0007↑.REST := ACTUALS;
```

```
ACTUALS := G0007;
COPYACTUALS := COPYTERMLIST(ACTUALS);
NEW(DONETBL);
DONETBL↑.NOTEMPTY := FALSE;
STANDAPART(COPYACTUALS, DONETBL);
NEW(MATCHLIST);
MATCHLIST↑.NOTEMPTY := FALSE;
NEW(G0014);
G0014↑.NOTEMPTY := TRUE;
NEW(G0015);
G0015↑.TTYP := CONSTANTTYP;
NEW(G0016);
G0016↑.CTYP := INTEGERTYP;
G0016↑.IVAL := 0;
G0015↑.CNST := G0016;
G0014↑.FIRST := G0015;
G0014↑.REST := MATCHLIST;
MATCHLIST := G0014;
NEW(G0011);
G0011↑.NOTEMPTY := TRUE;
NEW(G0012);
G0012↑.TTYP := CONSTANTTYP;
NEW(G0013);
G0013↑.CTYP := INTEGERTYP;
G0013↑.IVAL := 1;
G0012↑.CNST := G0013;
G0011↑.FIRST := G0012;
G0011↑.REST := MATCHLIST;
MATCHLIST := G0011;
IF UNIFY(COPYACTUALS , MATCHLIST,
          COPYACTUALS, MATCHLIST, FAILED)
 THEN BEGIN
      FAILED := NOT TRUE
     END
 ELSE FAILED := TRUE;
COPYACTUALS := COPYTERMLIST(ACTUALS);
NEW(DONETBL);
DONETBL↑.NOTEMPTY := FALSE;
STANDAPART(COPYACTUALS, DONETBL);
NEW(MATCHLIST);
MATCHLIST↑.NOTEMPTY := FALSE;
NEW(G0010);
G0010↑.NOTEMPTY := TRUE;
NEW(W);
W↑.TTYP := VARIABLE;
W↑.VR := GENVAR;
```

```
        G0010↑.FIRST := W;
        G0010↑.REST := MATCHLIST;
        MATCHLIST := G0010;
        NEW(G0009);
        G0009↑.NOTEMPTY := TRUE;
        NEW(Z);
        Z↑.TTYP := VARIABLE;
        Z↑.VR := GENVAR;
        G0009↑.FIRST := Z;
        G0009↑.REST := MATCHLIST;
        MATCHLIST := G0009;
        IF UNIFY(COPYACTUALS , MATCHLIST,
                 COPYACTUALS, MATCHLIST, FAILED)
         THEN BEGIN
              NEW(W1);
              W1↑.TTYP := VARIABLE;
              W1↑.VR := GENVAR;
              NEW(W1);
              W1↑.TTYP := VARIABLE;
              W1↑.VR := GENVAR;
              NEW(Z1);
              Z1↑.TTYP := VARIABLE;
              Z1↑.VR := GENVAR;
              NEW(Z1);
              Z1↑.TTYP := VARIABLE;
              Z1↑.VR := GENVAR;
              FAILED := NOT (SUB1 (W W1) AND FACT (W1 Z1)
                                 AND TIMES (W Z1 Z))
             END
         ELSE FAILED := TRUE;
        FLAG := NOT FAILED;
        FACT := FLAG;
        IF FLAG
         THEN BEGIN
              X :=COPYACTUALS↑.FIRST;
              COPYACTUALS := COPYACTUALS↑.REST;
              Y :=COPYACTUALS↑.FIRST;
              COPYACTUALS := COPYACTUALS↑.REST;
             END
      END
     ELSE FACT := FALSE
    END;
    BEGIN  END.
```

Appendix B: Specification of a Program Synthesis System

Note that *env* is used to denote a global environment containing information necessary to several of the following functions. The structure of *env* is a list of three elements: 1) a list of all generic function names along with their selection lists; 2) a list of the names of all functions defined so far; and 3) a list of all types defined so far. The environment, *env*, is initialized to a list consisting of three empty lists.

```
function syn
input pattern? (1 1 1 0 0)
parameter list?
syn(speci, target, envinit, program, newenv) ←
        firstsym(speci, nxtsym, spec),
        int(nxtsym, spec, envinit, int-prog, newenv),
        trans(int-prog, newenv, target, program).
```

```
function int
input pattern? (1 1 1 0 0)
parameter list?
int('function, spec, oldenv, int-prog, newenv) ←
        fun-spec(spec, oldenv, int-prog, newenv)
int('type, spec, oldenv, int-prog, newenv) ←
        type-spec(spec, oldenv, int-prog, newenv)
```

```
int('generic, spec, oldenv, int-prog, newenv) ←
        gen-spec(spec, oldenv, int-prog, newenv).


function fun-spec
input pattern? (1 1 0 0)
parameter list?
fun-spec(spec, oldenv, list('function, name, inpat, params, precond, postcond, body),
                newenv) ←
        firstsym(spec, name, more2),
        write("input-pattern?"), firstexp(more2, inpat, more3),
        write("parameter list?"), firstexp(more3, params, more4),
        write("precondition?"), is-disjunct(more4, precond, more5),
        write("postcondition?"), is-disjunct(more5, postcond, more6),
        addfun(oldenv, name, newenv),
        write("body?"), is-body(more6, body, moren, undef, newenv, inpat).


function is-disjunct
input pattern? (1 0 0)
parameter list?
is-disjunct(spec, disj, more) ←
        is-conjunct(spec, conj, more1, nxtsym),
        finish-disj(more1, nxtsym, conj, disj, more).


function finish-disj
input pattern?(1 1 1 0 0)
parameter list?
finish-disj(spec, 'v, conj, list('v, conj, disj), more) ←
        is-disjunct(spec, disj, more)
finish-disj(spec, 'f., conj, conj, spec).


function is-conjunct
input pattern? (1 0 0 0)
parameter list?
is-conjunct(spec, conj, more, nxtsym) ←
        firstsym(spec, litsym, more1),
        is-literal(litsym, more1, lit, more2, midsym),
        finish-conj(more2, midsym, lit, conj, more, nxtsym).


function finish-conj
```

```
input pattern? (1 1 1 0 0 0)
parameter list?
finish-conj(spec, '∧, lit, list('∧, lit, conj), more, nxtsym) ←
        is-conjunct(spec, conj, more, nxtsym)
finish-conj(spec, '/., lit, lit, spec, '/.).


function is-literal
input pattern? (1 1 0 0 0)
parameter list?
is-literal('TRUE, spec, T, more, nxtsym) ← firstsym(spec, nxtsym, more)
is-literal('T, spec, T, more, nxtsym) ← firstsym(spec, nxtsym, more)
is-literal('/(, spec, disj, more, nxtsym) ← is-disjunct(spec, disj, more1),
        firstsym(more1, '/), more2), firstsym(more2, nxtsym, more)
is-literal(name, spec, atmf, more, nxtsym) ← firstsym(spec, '/(, more1),
        is-funapp(name, more1, atmf, more, nxtsym).


function is-funapp
input pattern? (1 1 0 0 0)
parameter list?
is-funapp(name, spec, cons(name, arglist), more, nxtsym) ←
        is-arglist(spec, arglist, more, nxtsym).


function is-arglist
input pattern? (1 0 0 0)
parameter list?
is-arglist(spec, arglist, more, nxtsym) ←
        firstsym(spec, argsym, more2),
        read-args(argsym, more2, arglist, more,nxtsym).


function read-args
input pattern? (1 1 0 0 0)
parameter list?
read-args('/), spec, (), more, nxtsym) ← firstsym(spec, nxtsym, more)
read-args(argsym, spec, cons(arg,arglist), more, nxtsym) ←
        is-arg(argsym, spec, arg, more1, midsym),
        read-args(midsym, more1, arglist, more, nxtsym).


function is-arg
input pattern? (1 1 0 0 0)
```

```
parameter list?
is-arg(argsym, spec, const, more, nxtsym) ← is-constnt(argsym, spec, const, more1),
        firstsym(more1, nxtsym, more)
is-arg(name, spec, arg, more, lastsym) ← firstsym(spec, nxtsym, more1),
        finish-arg(nxtsym, name, more1, arg, more, lastsym).


function finish-arg
input pattern? (1 1 1 0 0 0)
parameter list?
finish-arg('/(, name, spec, fnapp, more, nxtsym) ←
        is-funapp(name, spec, fnapp, more, nxtsym)
finish-arg(nxtsym, name, spec, name, spec, nxtsym).


function is-constnt
input pattern? (1 1 0 0)
parameter list?
is-constnt('/', spec, list('quote,exp), more) ←
        firstexp(spec, exp, more)
is-constnt(number, spec, number, more) ← integer(number)
is-constnt(number, spec, number, more) ← real(number)
is-constnt('undef, spec, undef, more)
is-constnt('false, spec, false, more)
is-constnt('true, spec, t, more)
is-constnt('t, spec, t, more)
is-constnt('/(, spec, (), more) ← firstsym(spec,'/),more).


function is-body
input pattern? (1 0 0 1 1 1)
parameter list?
is-body(spec,cons('bktrkcond,alternatives),more,genflag,env,inpat) ←
        firstsym(spec, sym, more1),
        is-hornclauses(sym, more1,alternatives,more,genflag,env,inpat).


function is-hornclauses
input pattern? (1 1 0 0 1 1 1)
parameter list?
is-hornclauses('/., spec,(),more,genflag,env,inpat)
is-hornclauses(name, spec, cons(match-try-pair,alternatives),
                                more,genflag,env,inpat) ←
        firstsym(spec, '/(, spec2),
        is-hclause(name,spec2,match-try-pair,more1,genflag,env,inpat,nxtsym),
        is-hornclauses(nxtsym,more1,alternatives,more,genflag,env,inpat).
```

```
function is-hclause
input pattern? (1 1 0 0 1 1 1 0)
parameter list?
is-hclause(name,spec,list(arglist,trylist),more,genflag,env,inpat,lastsym) ←
        is-funapp(name,spec,cons(name,arglist),more1,nxtsym),
        finish-hclause(nxtsym, name, arglist, inpat, genflag, env, more1,
                                        trylist, lastsym).


function finish-hclause
input pattern? (1 1 1 1 1 1 1 0 0 0)
parameter list?
finish-hclause('←, name, arglist, inpat, genflag, env, spec, trylist,
                                                more, nxtsym) ←
        mk-known(inpat, arglist, (), knownvars),
        is-subgoalist(spec, trylist, more, env, knownvars, genflag, nxtsym)
finish-hclause(nxtsym, name, arglist, inpat, genflag, env, spec,
                        cons('try, ()), spec, nxtsym).


function is-subgoalist
input pattern? (1 0 0 1 1 1 0)
parameter list?
is-subgoalist(spec,cons('try,cons(cons(fname,arglist),sbglist)),more,env,
                knownvars,false,lastsym) ←
        firstsym(spec,name,spec2), firstsym(spec2, '|(, spec3),
        is-funapp(name,spec3,cons(name,arglist),more1,nxtsym),
        ck-generic(name,env,arglist, knownvars,fname),
        mk-allknown(arglist,knownvars,newknownvars),
        rd-subgoals(nxtsym,more1,sbglist,more,env,newknownvars,false,lastsym)
is-subgoalist(spec,cons('try,cons(cons(name,arglist),sbglist)),more,env,
                knownvars,true,lastsym) ←
        firstsym(spec,name,spec2), firstsym(spec2, '|(, spec3),
        is-funapp(name, spec3,cons(name,arglist),more1,nxtsym),
        rd-subgoals(nxtsym,more1,sbglist,more,env,knownvars,true,lastsym).


function rd-subgoals
input pattern? (1 1 0 0 1 1 1 0)
parameter list?
rd-subgoals('|,,spec,cons(cons(name,arglist),sbglist),
                        more,env,knownvars,true,lastsym) ←
        firstsym(spec,name,more1), firstsym( more1, '|(, more12),
        is-funapp(name,more12,cons(name,arglist),more2,nxtsym),
        rd-subgoals(nxtsym,more2,sbglist,more,env,knownvars,true,lastsym)
```

```
rd-subgoals('|,,spec,cons(cons(fname,arglist),sbglist),
                              more,env,knownvars,false,lastsym) ←
        firstsym(spec,name,more1), firstsym(more1, '|(, more21),
        is-funapp(name,more21,cons(name,arglist),more2,nxtsym),
        ck-generic(name,env,arglist,knownvars,fname),
        mk-allknown(arglist,knownvars,newknownvars),
        rd-subgoals(nxtsym,more2,sbglist,more,env,newknownvars,false,lastsym)
rd-subgoals(nxtsym, spec, (), spec, env, knownvars, genflag, nxtsym) .



function ck-generic
input pattern? (1 1 1 1 0)
parameter list?
ck-generic(name, env, arglist, knownvars, name) ← generic(name, env, undef)
ck-generic(name, env, arglist, knownvars, fname) ←
        generic(name, env, selections),
        mk_pat(arglist, knownvars, inpat),
        choose-fun(inpat, selections, fname).



function generic
input pattern? (1 1 0)
parameter list?
generic(name,list(generics,functions, types), selections) ←
        findin(name,generics,selections).



function findin
input pattern? (1 1 0)
parameter list?
findin(name, (), 'undef)
findin(name, cons(cons(name,x),y), x)
findin(name, cons(cons(other,x),y), z) ← findin(name,y,z).



function addfun
input pattern? (1 1 0)
parameter list?
addfun(list(generics,functions,types),name,
                list(generics, cons(name,functions),types)).
```

```
function not-in
input pattern? (1 1)
parameter list?
not-in(x,())
not-in(x,l) ← member(x,l,false).


function member
input pattern? (1 1 0)
parameter list?
member(x,cons(x,l),true)
member(x,cons(y,l),ans) ← member(x,l,ans)
member(x,(),false).


function mk-allknown
input pattern? (1 1 0)
parameter list?
mk-allknown((),knownvars,knownvars)
mk-allknown(cons(x,l), knownvars, newknownvars) ← vars_in(x,vl),
      mk-allknown(l, knownvars, nkv), append$(vl, nkv, newknownvars).


function mk-known
input pattern? (1 1 1 0)
parameter list?
mk-known((), (), knownvars, knownvars)
mk-known(cons(1,l), cons(x,k), knownvars, newknownvars) ← vars_in(x,vl),
      mk-known(l,k,knownvars, nkv), append$(vl, nkv, newknownvars)
mk-known(cons(0,l), cons(x,k), knownvars, newknownvars) ←
      mk-known(l, k, knownvars, newknownvars).


function vars_in
input pattern? (1 0)
parameter list?
vars_in(exp,()) ← itsaconstant(exp,true)
vars_in(exp, cons(exp,())) ← itsavar(exp,true)
vars_in(cons(name,arglist), varlist) ← varsinlist(arglist, varlist).
```

```
function varsinlist
input pattern? (1 0)
parameter list?
varsinlist((), ())
varsinlist(cons(s,l), varlist) ← vars_in(x, varlist1),
        varsinlist(l, varlist2), append$(varlist1, varlist2, varlist).


function itsaconstant
input pattern? (1 0)
parameter list?
itsaconstant(x, true) ← itsanumber(x, true)
itsaconstant(cons('quote,l), true)
itsaconstant(t, true)
itsaconstant('undef, true)
itsaconstant((), true)
itsaconstant('false, true)
itsaconstant('true, true)
itsaconstant('f, true)
itsaconstant(x, true) ← is-string(x).

function itsanumber
input pattern? (1 0)
parameter list?
itsanumber(x,true) ← real(x)
itsanumber(x, true) ← integer(x).

function itsavar
input pattern? (1 0)
parameter list?
itsavar(cons(x,y), false)
itsvar(exp, true) ← itsaconstant(exp, false).


function append$
input pattern? (1 1 0)
parameter list?
append$(x,(),x)
append$( (), x, x)
append$(cons(x,l1), l2, cons(x,l3)) ← append$(l1, l2, l3).


function choose-fun
input pattern? (1 1 0)
```

```
parameter list?
choose-fun(inpat, cons(pattern, cons(fname, sels)), fname) ←
        enuf-known(inpat, pattern, true)
choose-fun(inpat, cons(pattern, cons(fname,sels)), funname) ←
        choose-fun(inpat, sels, funname)
choose-fun(inpat, (), undef).
```

```
function enuf-known
input pattern? (1 1 0)
parameter list?
enuf-known((), (), true)
enuf-known(cons(1,l), cons(x,k), ans) ← enuf-known(l,k,ans)
enuf-known(cons(0,l), cons(1,k), false).
```

```
function mk_pat
input pattern? (1 1 0)
parameter list?
mk_pat( (), knownvars, ())
mk_pat(cons(arg,l), knownvars, cons(1,k)) ←
        is-known(arg, knownvars, true), mk_pat(l, knownvars, k)
mk_pat(cons(arg,l), knownvars, cons(0,k)) ← mk_pat(l, knownvars, k).
```

```
function is-known
input pattern? (1 1 0)
parameter list?
is-known(x, knownvars, true) ← itsaconstant(x,true)
is-known(cons(f,l), knownvars,ans) ← known-list(l, knownvars, ans)
is-known(x, knownvars, ans) ← member(x, knownvars, ans).
```

```
function knownlist
input pattern? (1 1 0)
parameter list?
known-list((), knownvars, true)
known-list(cons(x,l), knownvars, true) ← is-known(x, knownvars, true),
        known-list(l,knownvars,true).
```

```
function type-spec
input pattern? (1 1 0 0)
parameter list?
type-spec(spec,oldenv,
          list('type,name,'(1 0), '(x y), T, '(boolean y),body), newenv) ←
    firstsym(spec, name, more1),
    add-type(oldenv, name, newenv),
    write("body?"),
    is-body(more1, body, morex, undef, newenv, '(1 0)).


function add-type
input pattern? (1 1 0)
parameter list?
add-type(list(generics,functions,types), name,
                list(generics,functions, cons(name,types))).


function gen-spec
input pattern? (1 1 0 0)
parameter list?
gen-spec(spec, oldenv,
          cons(list('generic, name, params, selections), deflist), newenv) ←
    firstsym(spec, name, more1),
    write("parameter list?"), firstexp(more1, params, more2),
    write("choices?"), firstsym(more2, nxtsym, more3),
    rd-choices(nxtsym, more3, choicelist, bodylist, more4),
    add-gen(oldenv, name, choicelist, newenv),
    repeats-of(bodylist, rep-bodnams),
    write("body-defs:"),
    rd-bodies(more4, choicelist, (), (), params, rep-bodnams, newenv,
                deflist, morex).


function rd-choices
input pattern? (1 1 0 0 0)
parameter list?
rd-choices('!., spec, (), (), more)
rd-choices('!(, spec, cons(list(inpat,name,precond,postcond,bodnam), choicelist),
                cons(bodnam,bodylist), more) ←
    firstsym(spec, nxtsym, spec2),
    readinpat(nxtsym, spec2, inpat, more1), write("function name?"),
    firstsym(more1, name, more2), write("precondition?"),
```

```
        is-disjunct(more2, precond, more3),
        write("postcondition?"),
        is-disjunct(more3, postcond, more4),
        write("body name?"),
        firstsym(more4, bodnam, more5),
        write("choices?"), firstsym(more5, chsym, more6),
        rd-choices(chsym, more6, choicelist, bodylist, more).


function readinpat
input pattern? (1 1 0 0)
parameter list?
readinpat('/), spec, (), spec)
readinpat(digit, spec, cons(digit, restinpat), more) ←
        firstsym(spec, nxtsym, more1),
        readinpat(nxtsym, more1, restinpat, more).


function rd-bodies
input pattern? (1 1 1 1 1 1 1 0 0)
parameter list?
rd-bodies(spec, (), rep-bodies, donelist, params, rep-bodnams, env, (), spec)
rd-bodies(spec, cons(list(inpat,name,precond,postcond,bodnam), choicelist),
                rep-bodies, donelist, params, rep-bodnams,env,
                cons(list('function,name,inpat,params,precond,postcond,body), deflist),
                more) ←
        not-in(bodnam, donelist), not-in(bodnam,rep-bodnams),
        write(bodnam), write("?"),
        is-body(spec, body, more1, false, env, inpat),
        rd-bodies(more1, choicelist, rep-bodies, cons(bodnam,donelist), params,
                        rep-bodnams, env, deflist, more)
rd-bodies(spec, cons(list(inpat,name,precond,postcond,bodnam), choicelist),
                rep-bodies, donelist, params, rep-bodnams,env,
                cons(list('function,name,inpat,params,precond,postcond,body), deflist),
                more) ←
        member(bodnam,donelist,true),
        getb(bodnam, rep-bodies, genbody),
        spec-body(inpat, env, genbody, body)
rd-bodies(spec, cons(list(inpat,name,precond,postcond,bodnam), choicelist),
                cons(cons(bodnam,genbody),rep-bodies), donelist, params,
                        rep-bodnams,env,
                cons(list('function,name,inpat,params,precond,postcond,body), deflist),
                more) ←
```

```
        not-in(bodnam, donelist), member(bodnam, rep-bodnams, true),
        write(bodnam), write("?"),
        is-body(spec, genbody, more1, true, env, inpat),
        spec-body(inpat, env, genbody, body).


function getb
input pattern? (1 1 0)
parameter list?
getb(bodnam, cons(cons(bodnam,genbody), rep-bodies), genbody)
getb(bodnam, cons(x,repbodies), genbody) ←
        getb(bodnam, repbodies, genbody).


function spec-body
input pattern? (1 1 1 0)
parameter list?
spec-body(inpat, env, cons('bktrkcond, genalternatives),
                      cons('bktrkcond, alternatives) ) ←
        spec-alts(inpat, env, genalternatives, alternatives).


function spec-alts
input pattern? (1 1 1 0)
parameter list?
spec-alts(inpat, env, (), ())
spec-alts(inpat, env, cons(genmatch-try-pair, genalternatives),
                      cons(match-try-pair, alternatives)) ←
        spec-clause(inpat, env, genmatch-try-pair, match-try-pair),
        spec-alts(inpat, env, genalternatives, alternatives).


function spec-clause
input pattern? (1 1 1 0)
parameter list?
spec-clause(inpat, env, list(arglist, cons('try,genlist)),
                        list(arglist, cons('try,sblist))) ←
        mk-known(inpat, arglist, (), knownvars),
        spec-goalist(genlist, env, knownvars, sblist).
```

```
function spec-goalist
input pattern? (1 1 1 0)
parameter list?
spec-goalist((), env, knownvars, ())
spec-goalist(cons( cons(name,l), genlist), env, knownvars,
                    cons( cons(name,l),sblist)) ←
      generic(name, env, undef),
      mk-allknown(l, knownvars, newknownvars),
      spec-goalist(genlist, env, newknownvars, sblist)
spec-goalist(cons(cons(genname,arglist), genlist), env, knownvars,
                    cons(cons(name,arglist), sblist)) ←
      generic(genname, env, selections),
      mk_pat(arglist, knownvars, inpat),
      choose-fun(inpat, selections, name),
      mk-allknown(arglist,knownvars, newknownvars),
      spec-goalist(genlist, env, newkownvars, sblist).


function trans
input pattern? (1 1 1 0)
parameter list?
trans(list( 'function, name, inpat, params, precond, postcond,
                    cons('bktrkcond, alternatives)), env, 'lisp,
      list( 'defun, name, 'fexpr, '(l),
                    list( 'cond list( list('trueprecond,list('cons,
                                                    list('quote, name)
                                                    'l)),
                                          list('bktrkcond, 'l, list('quote,
                                                    alternatives))),
                                    '(t 'undef))) ).
```

Appendix C: Listing of the System

The following listing is the actual program that is read into MACLSP.
The semi-colon indicates that everything until the next carriage-return is to
be taken as a comment.

```
;top level
(def top ()
(prog (you-want-to-save)
; the next two lines should eventually be deleted, they make LISP the default language
(setq target 'lisp)
(makedefs (get 'lisp 'primdefs))
(setq namelist ())
(setq inflag t)
(setq prim-types '(integer real boolean is-string is-list))
(princ '|Hello, this is a program synthesis system which takes logic/
specifications as input and generates a program in the target/
language of your choice./
(Right now that choice is limited to LISP and maybe PASCAL).  /
/
If you wish the output to go to a file, please give me the name/
of that file; if not, just hit carriage-return/
|)

(readch)(readch)
((lambda (filename)
      (cond ((eq filename carriage_return) (setq outflag nil))
```

```
            (t (setq outputfile (read))
               (uwrite dsk (j red))
               (setq outflag t)
               (setq r t))))
 (tyipeek))

(princ '|/
If you wish me to read specifications from a file you've created,/
then please give me the name of the file; if this session is to be/
interactive, then just hit carriage-return/
|)
(readch)(readch)
((lambda (filename)
      (cond ((eq filename carriage_return) (setq inflag nil) (princ '|/
If you need help getting started, type "?"./
/
|))
            (t (setq filename (read))
               (setq inflag t)
               (eval (list 'eread filename))
               (setq q t))))
 (tyipeek))

(setq nxtsym (ratom))
(do () ((not (is_definition)) (print 'alldone))
      (setq namelist (cons name namelist))
      (putprop name params 'params)
      (putprop name precond 'precond)
      (putprop name inpat 'inpat)
      (putprop name postcond 'postcond)
      (putprop name body 'body)
      ((lambda (x) (cond ((eq target 'lisp)
                          (eval x) (eval (list 'grindef name)))
                         (t (unlist x))))
       (translate name target))
      (terpri)
      (terpri)
      (setq nxtsym (ratom)))
(cond (outflag
      (eval (list 'ufile outputfile 'gen))))
(princ '|/
Do you want to save these internalized specifications on a file?/
  If not, just hit carriage return... |)
(readch)
(readch)
(setq nxtsym (tyipeek))
```

```
(cond ((not (or (eq nxtsym carriage_return) (eq nxtsym $n)))
          (setq you-want-to-save t) (setq nxtsym (read)))) ;you-want-to-save
(cond (you-want-to-save (princ '|/
on what file?/ |)                                    ;is initialized to
                                                     ;nil as prog var
                  ((lambda (nam)
                        (dumpdefs namelist nam)
                        (princ '|/
Whenever you want to start the system up with these/
functions already defined as they were in this session./
type/
          "(include |)
                        (princ nam)(princ '|)"/
then type/
          "(top)"/
Actually the call on include can take any number of/
filenames that you wish to include.|))
                   (read)))
     (t (princ '|/
nothing saved/
|)))
(princ '|/
We now turn control back over to the top level of LISP. /
If you wish to start over type "(top)"/
|)))


(def is_definition  ()
      (do ()((not (eq nxtsym '?)))
                (princ '|/
To specify the target language type:/
          "target <language-name>"/
/
To specify a function definition type:/
          "function <name>"/
followed by the rest of the specification.  You will be asked for each/
part of the specification; if you don't know how to answer,/
type "?" for help./
/
To specify a generic function definition type:/
          "generic <name>"/
followed by the rest of the specification.  You will be asked for each/
part of the specification; if you don't know how to answer,/
type "?" for help./
/
To specify a data type type:/
```

```
        "type <name>"/
followed by the rest of the specification. You will be asked for each/
part of the specification; if you don't know how to answer,/
type "?" for help./
/
To conclude the session type a period "."/
|)
                                (terpri)
                                (setq nxtsym (ratom)))
        (setq genrlflag nil)
        (cond   ((is_fundef) (setq name fname)(putprop fname t 'function))
                ((is-gendef)(setq nxtsym (ratom)) (is_definition))
                ((is-typedef) (setq name typename)(putprop name t 'type))
                ((eq nxtsym 'target)(newtarget)(setq nxtsym (ratom))(is_definition))
                ((eq nxtsym '/.) nil)
                (t (error '(bad start (or finish)) nxtsym))))
(def newtarget ()
        (setq target (ratom))
        (cond ((eq target 'lisp)
                (setq deflist (get target 'primdefs))
                (makedefs deflist))))
(def makedefs (dl) (cond ((null dl) t)
                         (t (eval (first dl)) (makedefs (rest dl)))))

(def dumpdefs (namelist filename)
        (uwrite)
        (setq r t)
        (setq w t)
        (do ((names namelist (rest names)))
                ((null names))
                (prog (name)
                        (setq name (first names))
                        (print (list 'putprop (list 'quote name)
                                        (list 'quote (get name 'params))
                                        "params))
                        (print (list 'putprop (list 'quote name)
                                        (list 'quote (get name 'precond))
                                        "precond))
                        (print (list 'putprop (list 'quote name)
                                        (list 'quote (get name 'postcond))
                                        "postcond))
                        (print (list 'putprop (list 'quote name)
                                        (list 'quote (get name 'inpat))
                                        "inpat))
                        (print (list 'putprop (list 'quote name)
                                        (list 'quote (get name 'body))
```

```
                    "body))
                (print (list 'putprop (list 'quote name)
                             (list 'quote (get name 'fexpr))
                             "fexpr))
                ))      ;end of prog and do
        (print "*eof*)
        (eval (list 'ufile filename 'ext))
        (setq w nil))


(def include fexpr (flnames)
     (cond ((null flnames) 'all-done)
           (t  (eval (list 'eread (first flnames) 'ext))
               (setq q t)
               (do ((x nil (eval (read))))
                   ((eq x '*eof*)))
               (eval (cons 'include (rest flnames))))))


(def unlist (l) (do ((dumplist l (rest dumplist)))      ;this is for printing out
                    ((null dumplist))                   ;programs that were generated
                    (princ (first dumplist))))
```

```
;stuff needed all over

(putprop 'def (get 'defun 'fsubr) 'fsubr)
(setq $n 156)
(setq rpg-bug 315)
(setq ? 77)
(setq tab 11)
(setq period 56)
(setq dollar-sign 44)
(setq hot-cross-bun 26)
(setq comma 54)
(setq back-arrow 137)
(setq or-sym 37)
(setq and-sym 4)
(setq lpar 50)
(setq rpar 51)
(setq space 40)
(setq carriage_return 15)
(setq line_feed 12)

(def is_funapp ()
    (prog (fname arglist) (return
      (cond   ((eq (typep nxtsym) 'symbol) (setq fname nxtsym)
                                    (cond ((eq (setq nxtsym (ratom)) '/( )
                                          (setq nxtsym (ratom))
                                          (setq arglist (formalize
                                                        (readargs)))
                                          (make_funapp fname arglist))
                                         (t (error '(missing arglist) nxtsym))))
            (t (error '(funapp with bad function name) nxtsym))))))


;no infix function applications are allowed in this version


(def readargs ()
  (prog (arg)
        (cond ((eq nxtsym '/,) (setq nxtsym (ratom))))
  (return (cond ((eq nxtsym '/) ) (setq nxtsym (ratom)) ())
                ((eq (tyipeek) lpar) (cons (is_funapp) (readargs)))
                ((eq nxtsym '/() (cond ((eq (setq nxtsym (ratom)) '/))
                                       (setq arg ())
                                       (setq nxtsym (ratom))
                                       (cons arg (readargs)))
                                      (t (error '(unquoted non-empty list as arg)nxtsym))))
                ((atom nxtsym) (setq arg nxtsym) (setq nxtsym (ratom))
```

```
                                    (cons arg (readargs)))
                ((eq (first nxtsym) 'quote) (setq arg nxtsym) (setq nxtsym (ratom))
                                            (cons arg (readargs)))
                ((eq (first nxtsym) 'string) (setq arg nxtsym) (setq nxtsym (ratom))
                                            (cons arg (readargs)))
                (t (error '(weird argument) nxtsym)))))))


(def make_funapp (x y) (cons x y))
(def ratom () (setq nxtsym (tyipeek))
        (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                         (eq nxtsym tab) (eq nxtsym carriage_return))))
        (setq nxtsym (readch))
        (setq nxtsym (tyipeek)))
        (cond ((or (eq nxtsym comma) (eq nxtsym back-arrow) (eq nxtsym period)
                   (eq nxtsym or-sym) (eq nxtsym and-sym)
                   (eq nxtsym lpar) (eq nxtsym rpar)) (setq nxtsym (readch)))
              (t (setq nxtsym (read))
                 (cond ((eq nxtsym 'true) t)
                       ((eq nxtsym 'f) 'false)
                       (t nxtsym)))))
(def first (x) (car x))
(def second (x) (cadr x))


(def third (x) (caddr x))
(def rest (x) (cdr x))
```

```
;The following programs accomplish the interactive input of function, type,
;and generic definitions.  They are much longer than need be due to the
;voluminous help information.

(def is_fundef ()
  (cond ( (eq nxtsym 'function)
          (setq fname (read))
          (princ '|/
input pattern? |)
          (setq nxtsym (tyipeek))
          (do () ((not (eq nxtsym ?)))
              (setq nxtsym (readch))
              (princ '|/
|/
The input pattern is a list of 1's and 0's (optionally separated |/
by commas) indicating which of the parameters are to be considered|/
input (values available on procedure call) and which are output|/
(values to be computed)|/, respectively.  For example|/,|/
                  "(1|/, 1|/, 0)"|/
indicates that the last parameter is thought of as a function|/
of the first two parameters.|/
|/
input pattern? |)
                  (setq nxtsym (tyipeek)))
          (setq inpat (read))
          (princ '|/
parameter list?|/ |)
          (readch)(readch)
          (setq nxtsym (tyipeek))
          (do () ((not (or (eq nxtsym space) )))
              (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
          (do ()((not (eq nxtsym ?)))(setq nxtsym (readch))
              (princ '|/
the parameter list is a list of variables, enclosed in parentheses,|/
and optionally separated by commas. |/
                  For example, "(x1, x2, x3)" |)
                      (terpri)(princ '|/
parameter list?|/ |)
          (cond ((not inflag) (readch)(readch)))
                      (setq nxtsym (tyipeek))
          (do () ((not (or (eq nxtsym space) )))
              (setq nxtsym (readch)) (setq nxtsym (tyipeek))))
      (cond ((eq nxtsym carriage_return)
              (setq params (make-up-args inpat))
              (setq precond t)
              (setq postcond t)
```

```
                    (princ '|/
you've just defaulted on formal parameters, precondition,/
and postcondition, you know that really is not a good idea./
/
body? |))
            ((eq nxtsym lpar)
              (setq params (formalize (read)))
              (princ '|/
precond?/ |)
              (setq nxtsym (ratom))
              (do ()((not (eq nxtsym '?)))
                                  (princ '|/
a precondition is a disjunction, which expresses a condition /
or domain over which the function being defined is guaranteed/
to terminate. The disjunction must be terminated by a period.|) (terpri)
(princ '| /

domain-spec ::= disjunction "."/
/
disjunction ::= conjunction /| disjunction "∨" conjunction /
/
conjunction ::= literal /| conjunction "∧" literal /
/
literal ::= atomic-formula /| "¬" atomic-formula /
/
atomic-formula ::= "true" /| fun-app /| "(" disjunction ")" /
/
fun-app ::= name arglist /
/
arglist ::= "( )" /| "(" args ")"/
/
args ::= arg /| arg "," args /
/
arg ::= identifier /| number /| fun-app /
/
For example, "integer(x1,y) ∧ grtr-eq(x1,0,z)" /
Don't forget to include the output variables, in this case y and z.
|)
                                  (terpri)(terpri)(princ '|precond? |)
                                  (setq nxtsym (ratom)))
            (setq precond (readspec))
                  (princ '|/
postcond?/ |)
                  (setq nxtsym (ratom))
                  (do ()((not (eq nxtsym '?)))
                                  (princ '|/
```

```
A postcondition is a disjunction, which expresses a condition /
guaranteed to be true of the output variables.  It might also/
be considered as a specification of the range of the function.|)
(princ '| /
range-spec ::= disjunction "."/
/
disjunction ::= conjunction /| disjunction "∨" conjunction /
/
conjunction ::= literal /| conjunction "∧" literal /
/
literal ::= atomic-formula /| "¬" atomic-formula /
/
atomic-formula ::= "true" /| fun-app /| "(" disjunction ")" /
/
fun-app ::= name arglist /
/
arglist ::= "( )" /| "(" args ")"/
/
args ::= arg /| arg "," args /
/
arg ::= identifier /| number /| fun-app /
/
      For example, "integer(y,z) ∧ grtr(y,0,z)" /
(remember the output variables!)/
|)



                                (terpri)(princ '|/
postcond?/ |)
                                (setq nxtsym (ratom)))
            (setq postcond (readspec))
            (princ '|/
body?/ |))
          (t (error '(no "/(" seen when asking for parameters) nxtsym)))
            (setq nxtsym (ratom))
            (do ()((not (eq nxtsym '?)))
                (princ '|/
A function body is a set of horn clauses, terminated by a period./
/
body ::= horn-clauses "."/
/
horn-clauses ::= h-clause /| h-clause horn-clauses/
/
h-clause ::= goal "←" subgoals /| goal/
```

```
/
goal ::= fun-app/
/
subgoals ::= fun-app /| fun-app "," subgoals/
/
fun-app ::= name arglist /
/
For example:/
/
fact(0,1)/
fact(n,z) ← sub1(n,x), fact(x,z1), times(n,z1,z)/
|)
                                        (terpri)(princ '|/
body?/ |)
                                        (setq nxtsym (ratom)))
                            (setq body (repclauses)))
        (t nil)))


(def make-up-args (list)
        (cond ((null list) ())
              (t (cons (gensym) (make-up-args (rest list))))))


(def is-gendef ()
 (prog (specs gname params bodies-pair)
  (return
  (cond ( (eq nxtsym 'generic)
              (setq gname (read))
              (princ '|/
parameter list?/ |)
              (setq nxtsym (tyipeek))
              (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                               (eq nxtsym carriage_return))))
                     (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
              (do ()((not (eq nxtsym ?)))(setq nxtsym (readch))
                     (princ '|/
the parameter list is a list of variables, enclosed in parentheses,/
and optionally separated by commas. /
                            For example, "(x1, x2, x3)" |)
                                        (terpri)(princ '|/
parameter list?/ |)
                                        (setq nxtsym (tyipeek))
              (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                               (eq nxtsym carriage_return))))
                     (setq nxtsym (readch)) (setq nxtsym (tyipeek))))
```

```
            (setq params (formalize (read)))
               (princ '|
choices? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                             (eq nxtsym carriage_return))))
                (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
            (do ()((not (eq nxtsym ?)))(setq nxtsym (readch))
                (princ '|
a choice consists of an input pattern, function-name, precondition, postcondition,
and body-name. Just give the input pattern and the system will ask you for the
rest./
|
|
if there are no more choices to be entered type "."/
|
choices? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                             (eq nxtsym carriage_return))))
                (setq nxtsym (readch)) (setq nxtsym (tyipeek))))
      (prog (fun-name inpat precond postcond body-name)
          (setq specs
            (do ( (selections () (cons (cons inpat fun-name) selections))
                (flag () ())              ;flag tells whether fun-name is new
                (bodies () (cond (flag (cons body-name bodies))
                                 (t bodies)))
                (fun-names () (cond (flag
                                       (cons fun-name fun-names))
                                    (t fun-names))) )
               ( (eq nxtsym period)
                (setq nxtsym (ratom))
                (cons selections (cons bodies fun-names)) )
            (setq inpat (read))
            (princ '|
function name? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                             (eq nxtsym carriage_return))))
                (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
            (do ()((not (eq nxtsym ?)))(setq nxtsym (readch))
                (princ '|
this is the name the system will use to define a function with the given input
pattern, to be called whenever a generic call is made which fits the input-
pattern/
|
```

```
function name? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                             (eq nxtsym carriage_return))))
                (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
            (setq fun-name (read))
          (cond
            ((get fun-name 'body) nil);if it is already defined, do nothing
            ((eq '% (first (explode fun-name)))
                 (autopred fun-name params)); if system function, autopred it
            (t                              ;otherwise, get the rest of the info
             (setq flag t)
             (putprop fun-name inpat 'inpat)
             (putprop fun-name params 'params)
            (princ '|/
precond?/ |)
            (setq nxtsym (ratom))
            (cond ((eq nxtsym 'precond?) (setq nxtsym (ratom))))
            (do ()((not (eq nxtsym '?)))
                                (princ '|/
a precondition is a disjunction, which expresses a condition /
or domain over which the function being defined is guaranteed/
to terminate. The disjunction must be terminated by a period.|) (terpri)
(princ '| /

domain-spec ::= disjunction "."/
/
disjunction ::= conjunction /| disjunction "∨" conjunction /
/
conjunction ::= literal /| conjunction "∧" literal /
/
literal ::= atomic-formula /| "¬" atomic-formula /
/
atomic-formula ::= "true" /| fun-app /| "(" disjunction ")" /
/
fun-app ::= name arglist /
/
arglist ::= "( )" /| "(" args ")"/
/
args ::= arg /| arg "," args /
/
arg ::= identifier /| number /| fun-app /
/
For example, "integer(x1,y) ∧ grtr-eq(x1,0,z)" /
Don't forget to include the output variables, in this case y and z.
|)
```

```
                                (terpri)(terpri)(princ 'precond? )
                                (setq nxtsym (ratom))
                                (cond ((eq nxtsym 'precond?) (setq nxtsym
                                                              (ratom)))))
            (setq precond (readspec)) (putprop fun-name precond 'precond)
                (princ '|/
postcond?/ |)
                (setq nxtsym (ratom))
                (cond ((eq nxtsym 'postcond?) (setq nxtsym (ratom))))
                (do ()(not (eq nxtsym '?)))
                                (princ '|/
A postcondition is a disjunction, which expresses a condition /
guaranteed to be true of the output variables.  It might also/
be considered as a specification of the range of the function.|)
(princ '| /
range-spec ::= disjunction "."/
/
disjunction ::= conjunction /| disjunction "∨" conjunction /
/
conjunction ::= literal /| conjunction "∧" literal /
/
literal ::= atomic-formula /| "¬" atomic-formula /
/
atomic-formula ::= "true" /| fun-app /| "(" disjunction ")" /
/
fun-app ::= name arglist /
/
arglist ::= "( )" /| "(" args ")"/
/
args ::= arg /| arg "," args /
/
arg ::= identifier /| number /| fun-app /
/
      For example, "integer(y,z) ∧ grtr(y,0,z)" /
(remember the output variables!)/
|)



                                (terpri)(princ '|/
postcond?/ |)
                                (setq nxtsym (ratom))
                                (cond ((eq nxtsym 'postcond?)
                                                (setq nxtsym (ratom)))))
            (setq postcond (readspec)) (putprop fun-name postcond 'postcond)
```

```
            (princ '|/
body-name? |)
            (cond ((not inflag) (ratom)))
            (setq nxtsym (ratom))
            (do ()((not (eq nxtsym '?)))
                  (princ '|/
this is the name which associates the proper body definition with the function
being defined/
/
body-name?/ |)
                  (cond ((not inflag) (ratom)))
                  (setq nxtsym (ratom)))
                  (setq body-name nxtsym)
                  (putprop fun-name body-name 'bodyname)
                              ));end of cond for defining fun-name
               (princ '|/
choices? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                        (eq nxtsym carriage_return))))
                  (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
            (do ()((not (eq nxtsym ?)))(setq nxtsym (readch))
                  (princ '|/
a choice consists of an input pattern, function-name, precondition, postcondition,
and body-name. Just give the input pattern and the system will ask you for the
rest./
/
if there are no more choices to be entered type "."/
/
choices? |)
            (setq nxtsym (tyipeek))
            (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                        (eq nxtsym carriage_return))))
                  (setq nxtsym (readch)) (setq nxtsym (tyipeek))))
                              )))
;end of the "do" that gets all the choices and the setq of the result of "do"
;and the prog surrounding it
            (print (make-gendef gname (reverse (first specs))))
      (cond ((first (rest specs))
            (setq bodies-pair (splitbodies (second specs) '(()())) )
            (setq namelist (append (rest (rest specs)) namelist))
            (princ '|/
body-defs:|)
        (prog (bodnam fun-name genrlflag genrldef)
            (do ( (rep-bodies (second bodies-pair)  rep-bodies)
                (donebods () (cons bodnam donebods))
```

```
                (fun-names (reverse (rest (rest specs))) (rest fun-names))
                (rep-defs () (cond ((and genrlflag (not (member
                                                    bodnam donebods)))
                                    (cons (cons bodnam genrldef) rep-defs))
                                   (t rep-defs))) )
               ((null fun-names))
               (setq fun-name (first fun-names))
               (setq bodnam (get fun-name 'bodyname))
               (setq genrlflag (member bodnam rep-bodies))
               (cond ((member bodnam donebods) (spec-def fun-name
                                                 (getdef bodnam rep-defs)))
                     (t
                      (terpri)
                      (princ bodnam)
                      (princ '|? |)
                      (setq nxtsym (tyipeek))
                      (do () ((not (or (eq nxtsym space) (eq nxtsym line_feed)
                                       (eq nxtsym carriage_return))))
                          (setq nxtsym (readch)) (setq nxtsym (tyipeek)))
                      (setq nxtsym (ratom))
                      (do ()((not (eq nxtsym '?)))
                          (princ '|/
A function body is a set of horn clauses, terminated by a period./
/
body ::= horn-clauses "."/
/
horn-clauses ::= h-clause /| h-clause horn-clauses/
/
h-clause ::= goal "←" subgoals /| goal/
/
goal ::= fun-app/
/
subgoals ::= fun-app /| fun-app "," subgoals/
/
fun-app ::= name arglist /
/
For example:/
/
fact(0,1)/
fact(n,z) ← sub1(n,x), fact(x,z1), times(n,z1,z)/
|)
                                  (terpri)
                                  (princ bodnam)
                                  (princ '|? |)
                                  (setq nxtsym (ratom))
                                  (cond ((eq nxtsym (implode (append
```

```
                                                    (explode bodnam)
                                                    '(?)))))
                                          (setq nxtsym (ratom)))))
                                (setq inpat (get fun-name 'inpat))
                                (putprop fun-name
                                  (cond (genrlflag
                                          (setq genrldef (repclauses))
                                          (spec-def fun-name genrldef))
                                        (t (repclauses)))
                                  'body) )))) ;end of do collecting body-defs
                        (make-fundefs (rest (rest specs))))) ;end of conditional on bodies
                        (print 'end_of_generic_spec) (terpri) (terpri)
                                    t) ;end first clause of cond
                (t nil)))))


(def make-gendef (name  sel-list)
      (putprop name sel-list 'generic))

(def make-fundefs (fun-names)
      (cond
         ((null fun-names) t)
         (t
            ((lambda (x) (eval x)
                     (eval (list 'grindef (first fun-names))))
              (translate (first fun-names) target))
            (putprop (first fun-names) t 'function)
            (terpri) (terpri)
            (make-fundefs (rest fun-names)))))

(def splitbodies (bods ans)
      (cond ((null bods) ans)
            (t (splitbodies (rest bods)
                        (checkbod (first bods) ans)))))

(def checkbod (bodnam uniq-n-repeats)
      (cond ((member bodnam (first uniq-n-repeats))
             (list (first uniq-n-repeats)
                   (cons bodnam (second uniq-n-repeats))))
            (t (list (cons bodnam (first uniq-n-repeats))
                     (second uniq-n-repeats)))))

(def getdef (name deflist)
      (cond ((null deflist) (error
                        '(looking for a genrl definition that does no exist) name))
            ((eq name (first (first deflist)))
```

```
            (rest (first deflist)))
           (t (getdef name (rest deflist)))))

(def spec-def (fun-name genrldef)
     (cons 'bktrkcond (spec-altlist (rest genrldef) (get fun-name 'inpat))))

(def spec-altlist (genalts inpat)
     (cond ((null genalts) ())
          (t ((lambda (matchlist)
               (setq knownvars (known-of matchlist inpat))
               (cons (list matchlist
                         (cons 'try (spec-goals (goal-pt (first genalts)))))
                     (spec-altlist (rest genalts) inpat)))
             (match-pt (first genalts))))))

(def goal-pt (alternative) (rest (second alternative)))

(def spec-goals (gengoals)
     (prog (tempgoal choices)
       (return (cond ((null gengoals) ())
                     ((setq choices
                          (is-generic (setq tempgoal (first gengoals))))
                      (cons (choosefun tempgoal choices)
                          (spec-goals (rest gengoals))))
                     (t (cons tempgoal (spec-goals (rest gengoals))))))))

(def formalize (arglist)
     (cond ((null arglist) ())
          ((is-var (first arglist))
               (cons (implode (cons '! (explode (first arglist))))
                    (formalize (rest arglist))))
          (t  (cons (first arglist) (formalize (rest arglist))))))


(def is-typedef ()
  (cond ((eq nxtsym 'type)
         (setq typename (read))
         (terpri)
         (setq params '(!x !y))
         (setq inpat '(1 0))
         (setq precond t)
         (setq postcond '(boolean !y t))
         (princ '|/
body?/ |)
         (setq nxtsym (ratom))
         (do () ((not (eq nxtsym '?)))
```

```
                    (princ '|/
For a type definition, the input pattern is always (1 0), the/
precondition is true, and the postcondition is that the output/
variable will have a boolean value./
/
A type body is a set of horn clauses,terminated by a/
period; it can be considered as the definition of a/
function to test for membership in the type.  It /
always has a single input variable, which can be /
anything, and a single output variable, which is /
always truth-valued./
/
body ::= horn-clauses "."
/
horn-clauses ::= h-clause /| h-clause horn-clauses/
/
h-clause ::= goal "←" subgoals /| goal/
/
goal ::= fun-app/
/
subgoals ::= fun-app /| fun-app "," subgoals/
/
fun-app ::= name arglist /
/
For example:/
tree(empty-tree,y)/
tree( graft(t1,t2), y) ← tree(t1,y1), tree(t2,y2)/
|)
                                (terpri)(princ '|/
body?/ |)
                                (setq nxtsym (ratom)))
          (setq body (repclauses)))
        (t nil)))
(def put-types (l)
       (cond ((null l) t)
             (t (putprop (first l) typename 'typename)
                (put-types (rest l)))))
```

```
;The following programs are for reading in precondition and postcondition
;specifications

(def readspec ()
     (prog (disj)
       (return (cond ((setq disj (is_disjunct)) disj)
            (t (error (specification is not disjunction) nxtsym))))))

(def is_disjunct ()
    (prog (conj1 conj2) (return
      (cond   ((setq conj1 (is_conjunct))
                  (do () ((not (eq nxtsym 'v)) conj1)
                        (setq nxtsym (ratom))
                        (cond ((setq conj2 (is_conjunct))
                                    (setq conj1 (make_or conj1 conj2)))
                              (t (error '(v not followed by conjunct) nxtsym)))))
            (t (error '(no conjunct to start with) nxtsym))))))

(def make_or (x y) (list 'v x y))
(def make_and (x y) (list '^ x y))
(def is_conjunct ()
    (prog (lit1 lit2) (return
      (cond ((setq lit1 (is_literal))
                  (do () ((not (eq nxtsym '^)) lit1)
                        (setq nxtsym (ratom))
                        (cond ((setq lit2 (is_literal))
                                    (setq lit1 (make_and lit1 lit2)))
                              (t (error '(^ not followed by literal) nxtsym)))))
            (t (error '(no literal to start with) nxtsym))))))
(def is_literal ()
    (prog (atmf) (return
      (cond   ((or (eq nxtsym t) (eq nxtsym 'true)) (setq nxtsym (ratom)) t)
            ((eq nxtsym '( ) (setq nxtsym (ratom))
                        (cond ((setq atmf (is_disjunct))
                                    (cond ((eq nxtsym ') )
                                          (setq nxtsym (ratom))
                                          atmf)
                                          (t (error '(missing right paren) nxtsym))))
                              (t (error '(no disjunct in parens) nxtsym))))
            (t (is_funapp))))))
```

```
;The following programs read in and internalize Horn clauses

(def repclauses ()
       (cons 'bktrkcond (read_alternatives)))
;      (cond ((reg-cond altlist) (cons 'cond altlist))
;            (t (cons 'bktrkcond altlist))))

(def read_alternatives ()
       (cond ((eq nxtsym '/.) ())
             (t (cons (is_alternative) (read_alternatives)))))

;"←" should only appear in clauses which have something to the right of it
(def is_alternative ()
       (do ()(not (eq nxtsym '?)))
                          (princ '|/
You are in the midst of specifying the body, a horn clause is looked for./
Don't forget that a single-goal clause does NOT have "←" following the goal,/
and the commas which separate subgoals are mandatory./
Here is a grammar description:/
/
horn-clauses ::= h-clause /| h-clause horn-clauses/
/
h-clause ::= goal "←" subgoals /| goal/
/
goal ::= fun-app/
/
subgoals ::= fun-app /| fun-app "," subgoals/
/
fun-app ::= name arglist /
/
|)
                          (terpri)(princ '|/
                                   horn-clause?/ |)
                          (setq nxtsym (ratom)))
      (cond ((setq goal (is_funapp))
                 (setq known-vars (known-of (rest goal) inpat))
                 (list (rest goal) (make_try (list_subgoals))))
            (t (error '(bad goal) nxtsym))))


(def known-of (termlist pattern)
      (cond   ((null termlist) () )
              ((eq (first pattern) 1) (append (vars-in (first termlist))
                                              (known-of (rest termlist)
                                                        (rest pattern))))
              (t (known-of (rest termlist) (rest pattern)))))
```

```
(def vars-in (term)
     (cond ( (atom term)
               (cond ((is-var term)
                      (list term))
                     (t () )) )
           ( t (vars-in-list (rest term)))))


(def vars-in-list (termlist)
     (cond   ((null termlist) () )
             (t (append (vars-in (first termlist))
                        (vars-in-list (rest termlist))))))



(def list_subgoals ()
   (prog (tempgoal choices)
      (return (cond ((eq nxtsym '←)
                     (setq nxtsym (ratom))
                     (setq tempgoal (is_funapp))
                     (cond ((and (not genrlflag)
                                 (setq choices (is-generic tempgoal)))
                            (cons (choosefun tempgoal choices)(read_subgoals)))
                           ((eq '% (first (explode (first tempgoal))))
                                 (autopred (first tempgoal) (rest tempgoal))
                                 (addoutvars tempgoal)
                                 (cons tempgoal (read_subgoals)))
                           ((not (null tempgoal))
                              (addoutvars tempgoal)
                              (cons tempgoal (read_subgoals)))
                           (t (error '(subgoal is not a funapp) nxtsym))))
                    (t ()))))

(def is-generic (call) (get (first call) 'generic))

(def addoutvars (call)
     (do ((pat (get (first call) 'inpat) (rest pat))
          (varlist (rest call) (rest varlist)))
         ((null pat))
         (cond ((eq (first pat) 0)
                (setq known-vars (append (vars-in (first varlist))
                                         known-vars))))))


(def choosefun (call choicelist)
     (prog (pat fun)
```

```
                (setq pat (mk-pat (rest call)))
                (setq fun (findfun pat choicelist))
                (addoutvars (cons fun (rest call)))
                (return (cons fun (rest call)))))

(def mk-pat (varlist)
      (cond ((null varlist) () )
            ((is-constant (first varlist))
              (cons 1 (mk-pat (rest varlist))))
            ((is-var (first varlist))
              (cond ((memq (first varlist) known-vars)
                           (cons 1 (mk-pat (rest varlist))))
                    (t (cons 0 (mk-pat (rest varlist))))))
            ((all-vars-known (first varlist))
              (cons 1 (mk-pat (rest varlist))))
            (t (cons 0 (mk-pat (rest varlist))))))

(def findfun (key pairlist)
      (cond   ((null pairlist)
                   (princ '| /
i can't figure out which function you want from context, please help./
the patterns and function names you've given me are:/
|) (princ choicelist) (princ '|/
known-vars is: |) (princ known-vars) (princ '|/
which function do you want? |)
                   (read))
              ((as-defined-as key (first (first pairlist)))
                   (rest (first pairlist)))
              (t (findfun key (rest pairlist)))))


(def as-defined-as (pat1 pat2)
      (cond ((null pat1) t)
            ((and (eq (first pat1) 0) (eq (first pat2) 1)) nil)
            (t (as-defined-as (rest pat1) (rest pat2)))) .



(def all-vars-known (exp)
      (cond ((atom exp) (or (is-constant exp)
                            (memq exp known-vars)))
            (t (all-vars-in-list-known (rest exp)))))

(def all-vars-in-list-known (l)
      (cond ((null l) t)
            ((all-vars-known (first l))
```

```
                (all-vars-in-list-known (rest l)))
               (t nil)))


(def make_match (x y) (list 'match x y))

(def make_try (l) (cons 'try l))
(def read_subgoals ()
  (prog (tempgoal choices)
   (return (cond ((eq nxtsym '/,)
                  (setq nxtsym (ratom))
                  (setq tempgoal (is_funapp))
                  (cond ((and (not genrlflag)
                              (setq choices (is-generic tempgoal)))
                         (cons (choosefun tempgoal choices)(read_subgoals)))
                        ((eq '% (first (explode (first tempgoal))))
                              (autopred (first tempgoal) (rest tempgoal))
                              (addoutvars tempgoal)
                              (cons tempgoal (read_subgoals)))
                        ((not (null tempgoal))
                         (addoutvars tempgoal)
                         (cons tempgoal (read_subgoals)))
                        (t (error '(subgoal is not a funapp) nxtsym))))
                 (t ())))))
(def is-constant (x)
      (cond ((atom x) (or (numberp x)
                          (eq x t)
                          (eq x 'f)
                          (eq x 'true)
                          (eq x 'undef)
                          (eq x 'false)
                          (eq x nil)))
            (t (or (not (is-string x)) (eq (first x) 'quote)
                   (not (contains-var x)))))))

(def is-var (x) (and (atom x) (not (is-constant x))))

(def contains-var (exp)
      (cond ((atom exp) (is-var exp))
            (t (list-contains-var (rest exp)))))
;contains-var ignores function names when looking for variables since the only
;functions left in at this point are constructors
(def list-contains-var (explist)
      (cond ((null explist) nil)
            ((contains-var (first explist)) t)
```

```
(t (list-contains-var (rest explist)))))
```

```
;The following programs perform the translation to target language

(def translate (name target)
     (cond  ((eq target 'lisp)
             (make_lisp_def))
            ((eq target 'pascal)
                 (mk-strong-typed)
                 (make_pascal_def))
            (t (error '(language not yet implemented) target))))

(def make_lisp_def () (list 'defun name 'fexpr '(l)
                  (list 'cond (list
                                (list 'true-precond (list 'cons
                                                      (list 'quote
                                                          name)
                                                      'l))
                                (list 'bktrkcond
                                      'l
                                      (list 'quote (rest body))))
                              '(t 'undef))))


(def mk-strong-typed ()
     (make-formal-types (get name 'params)  ;this function puts the formal params
                (get name 'precond)     ;and their types under 'types, and
                (get name 'postcond)    ;deletes the type decs from the pre-
                name)                   ;and post-conditions, putting the
                                        ;results under 'typedprecond and
                                        ;typedpostcond

     (putprop name (findprocs (rest (get name 'body)) name) 'external-procs)
     (putprop name (find-local-types (rest (get name 'body)) name) 'local-decs))


(def make-formal-types (params precond postcond name)
     ((lambda (split-pre split-post)
          (putprop name (mk-param-types params
                              (append (first split-pre)
                                   (first split-post)))
                   'types)
          (putprop name (rest split-pre) 'typedprecond)
          (putprop name (rest split-post) 'typedpostcond))
      (findtypes precond)
      (findtypes postcond)))
```

```
(def mk-param-types (params types)
     (cond ((null params) ())
           (t ((lambda (ans)
                  (cond ((defined ans)
                         (cons ans (mk-param-types (rest params)
                                                  types)))
                        (t (error '(has no type defined) (first params)))))
               (lookup (first params) types)))))


(def lookup (var alist)
     (cond   ((null alist) 'undef)
             ((eq var (first (first alist)))
                 (rest (first alist)))
             (t (lookup var (rest alist)))))


(def findtypes (precond)        ;returns the dotted-pair: list of var-type pairs,
     (cond ((atom precond)          ;and non-type-dec part of precond
                (cons () precond))
           ((is-or precond)
                (error '(no v's allowed in preconditions when translating
                           to strongly typed languages) precond))
            ;if we switch from dnf to cnf, then we can allow v's to appear
            ;in pre- and post-conditions, but no type decs may be v'ed, the
            ;alternative replacing the error above would be:
            ;(cons () precond)
           ((is-and precond)
                ((lambda (rest-ans1 rest-ans2)
                     (cons (append (first rest-ans1)
                                   (first rest-ans2))
                           (list '^ (rest rest-ans1)
                                    (rest rest-ans2))))
                 (findtypes (second precond))
                 (findtypes (third precond))))
           ((is-type (first precond))                    ;we know that the first
                (cons (list (cons (second precond)       ;argument of a type
                                  (first precond)))      ;app. is the input var
                      T))))


(def is-type (name)
     (or (get name 'type)
         (memq name prim-types)))
```

```
(def findprocs (alts name)
     (cond ((null alts) ())
          (t (append (findcalls (try-pt (first alts)) name)
                     (findprocs (rest alts) name)))))


(def findcalls (sbgls name)
     (cond ((null sbgls) ())
          (t ((lambda (procname)
                   (cond ((eq procname name)
                              (findcalls (rest sbgls) name))
                         (t (cons procname (findcalls (rest sbgls) name)))))
              (first (first sbgls))))))


(def find-local-types (alts name)
     (cond ((null alts) ())
          (t (append (findtypesfor (try-pt (first alts)) name)
                     (find-local-types (rest alts) name)))))


(def findtypesfor (sbgls name)
     (cond ((null sbgls) ())
          (t ((lambda (sbgl)
                   (cond ((eq (first sbgl) name)
                              (findtypesfor (rest sbgls) name))
                         (t (append (make-type-list (rest sbgl)
                                                    (get (first sbgl) 'types))
                                    (findtypesfor (rest sbgls) name)))))
              (first sbgls)))))


(def make-type-list (actuals type-pattern)      ;returns a list of var-type pairs
     (cond ((nul actuals) ())
          ((is-var (first actuals))
                   (cons (cons (first actuals)
                               (first type-pattern))
                         (make-type-list (rest actuals)
                                         (rest type-pattern))))
          ((is-constant (first actuals))
                   (make-type-list (rest actuals)
                                   (rest type-pattern)))
          (t (append                ;it's a funapp, i.e., a constructed type, so
                   (hard-type-list (first actuals) ;lookup def of type and make
                                   (first type-pattern))  ;sure the constructor
                   (make-type-list (rest actuals)  ;is appropriate, then find
```

```
                                (rest type-pattern)))))) ;types of the args


(def hard-type-list (exp type)
      (cond ((eq (vars-in exp) ()) ())
            (t ((lambda (bod)
                  (cond (bod (srch-alts (list exp (gensym))   ;assumes inpat (1 0)
                                        (rest bod)))
                        (t (error '(has not yet been defined) type))))
                (get type 'body))))))


(def srch-alts (actuals list-alts)
      (cond ((null list-alts) (error '(has vars whose types I am unable to determine)
                                     (first actuals)))
            (t ((lambda (sbgls)
                  (cond ((defined sbgls)
                         ((lambda (typedecs)
                            (cond ((null typedecs)
                                   (srch-alts actuals
                                              (rest list-alts)))
                                  (t ((lambda (ans)
                                        (cond ((all-typed
                                                (vars-in (first actuals))
                                                ans)
                                               ans)
                                              (t (srch-alts actuals
                                                            (rest list-alts)))))
                                      (find-var-types
                                        (vars-in (first actuals))
                                        (append (first sbgls)
                                                typedecs))))))
                          (find-type-decs (rest sbgls))))
                        (t (srch-alts actuals (rest list-alts)))))
                (half-eval actuals (first list-alts))))))


(def half-eval (actuals alt)   ;returns a substitution and list of subgoals with the
      ((lambda (sub)           ;substitutions made
            (cond ((defined sub)
                   (cons sub (mk-subst (try-pt alt) sub)))
                  (t 'undef)))
       (match actuals (match-pt alt))))


(def find-type-decs (sbgls)
```

```
        (cond ((null sbgls) ())
              (t ((lambda (sbgl)
                          (cond ((is-type (first sbgl))
                                 (append (make-type-list (list (second sbgl))
                                                         (list (first sbgl)))
                                         (find-type-decs (rest sbgls))))
                                (t (find-type-decs (rest sbgls)))))
                  (first sbgls)))))


(def find-var-types (vars sub)
     (cond ((null vars) ())
           (t (cons (cons (first vars)
                          (lookup* (first vars) sub))
                    (find-var-types (rest vars) sub)))))


(def all-typed (vars alist)
     (cond ((null vars) t)
           ((is-type (lookup (first vars) alist))
                     (all-typed (rest vars) alist))
           (t nil)))
```

16.1 Listing of LISP Implementation

```
;primitive function and type definitions
;for LISP

(PROG2
(putprop 'lisp '( (def integer fexpr (l)
                    (cond ((eq (length l) 1)
                           (cond ((fixp (first l)) ())
                                 (t 'undef)))
                          ((is-constant (second l))
                           (cond ((fixp (first l))
                                  (cond ((or (eq (second l) 'true)
                                             (eq (second l) t)) () )
                                        (t 'undef)))
                                 ((or (eq (second l) 'undef)
                                      (eq (second l) 'false)) ())
                                 (t 'undef)))
                          (t (list (cons (second l)
                                         (cond ((fixp (first l)) t)
                                               (t 'false)))))))
                  (def real fexpr (l)
                       (cond ((eq (length l) 1)
                              (cond ((floatp (first l)) ())
                                    (t 'undef)))
                             ((is-constant (second l))
                              (cond ((floatp (first l))
                                     (cond ((or (eq (second l) 'true)
                                                (eq (second l) t)) () )
                                           (t 'undef)))
                                    ((or (eq (second l) 'undef)
                                         (eq (second l) 'false)) ())
                                    (t 'undef)))
                             (t (list (cons (second l)
                                            (cond ((floatp (first l)) t)
                                                  (t 'false)))))))
                  (def boolean fexpr (l)
                       (cond ((eq (length l) 1)
                              (cond ((or (eq (first l) t)
                                         (eq (first l) 'true)
                                         (eq (first l) 'undef)
                                         (eq (first l) 'false)) ())
                                    (t 'undef)))
                             ((is-constant (second l))
                              (cond ((or (eq (first l) t)
                                         (eq (first l) 'true)
```

```
                    (eq (first l) 'undef)
                    (eq (first l) 'false))
                  (cond ((or (eq (second l) 'true)
                             (eq (second l) t)) () )
                        (t 'undef)))
                 ((or (eq (second l) 'undef)
                      (eq (second l) 'false)) ())
                 (t 'undef)))
           (t (list (cons (second l)
                          (cond
                           ((or (eq (first l) t)
                                (eq (first l) 'true)
                                (eq (first l) 'undef)
                                (eq (first l) 'false)) t)
                           (t 'false)))))))
(def ≤ fexpr (l)
     (cond ((eq (length l) 2)
            (cond ((or (old< (first l) (second l))
                       (old= (first l) (second l))) ())
                  (t 'undef)))
           ((is-constant (third l))
            (cond ((or (old< (first l) (second l))
                       (old= (first l) (second l)))
                   (cond ((or (eq (third l) t)
                              (eq (third l) 'true)) ())
                         (t 'undef)))
                  ((or (eq (third l) 'false)
                       (eq (third l) 'undef))())
                  (t 'undef)))
           (t (list (cons (third l)
                          (cond ((or (old< (first l) (second l))
                                     (old= (first l) (second l)))
                                 t)
                                (t 'false)))))))
(def ≥ fexpr (l)
     (cond ((eq (length l) 2)
            (cond ((or (old> (first l) (second l))
                       (old= (first l) (second l))) ())
                  (t 'undef)))
           ((is-constant (third l))
            (cond ((or (old> (first l) (second l))
                       (old= (first l) (second l)))
                   (cond ((or (eq (third l) t)
                              (eq (third l) 'true)) ())
                         (t 'undef)))
                  ((or (eq (third l) 'false)
```

```
                    (eq (third l) 'undef))())
                (t 'undef)))
            (t (list (cons (third l)
                    (cond ((or (old> (first l) (second l))
                               (old= (first l) (second l)))
                           t)
                          (t 'false)))))))
(def ≠ fexpr (l)
    (cond ((eq (length l) 2)
           (cond ((old= (first l) (second l)) 'undef)
                 (t ())))
          ((is-constant (third l))
           (cond ((old= (first l) (second l))
                  (cond ((or (eq (third l) 'undef)
                             (eq (third l) 'false)) ())
                        (t 'undef)))
                 ((or (eq (third l) 'true)
                      (eq (third l) t))())
                 (t 'undef)))
          (t (list (cons (third l)
                   (cond ((old= (first l) (second l))
                          'false)
                         (t t)))))))
(putprop 'old< (get '< 'subr) 'subr)
(putprop 'old> (get '> 'subr) 'subr)
(putprop 'old= (get '= 'subr) 'subr)
(def < fexpr (l)
    (cond ((eq (length l) 2)
           (cond ((old< (first l) (second l)) ())
                 (t 'undef)))
          ((is-constant (third l))
           (cond ((old< (first l) (second l))
                  (cond ((or (eq (third l) 'true)
                             (eq (third l) t)) ())
                        (t 'undef)))
                 ((or (eq (third l) 'false)
                      (eq (third l) 'undef)) ())
                 (t 'undef)))
          (t (list (cons (third l)
                   (cond ((old< (first l) (second l)) t)
                         (t 'false)))))))
(def v fexpr (l)
    (cond ((eq (length l) 2)
           (cond ((or (first l) (second l)) ())
                 (t 'undef)))
          ((is-constant (third l))
```

```
                    (cond ((or (first l) (second l))
                           (cond ((or (eq (third l) 'true)
                                      (eq (third l) t)) ())
                                 (t 'undef)))
                          ((or (eq (third l) 'undef)
                               (eq (third l) 'false)) ())
                          (t 'undef)))
                (t (list (cons (third l)
                               (cond ((or (first l) (second l)) t)
                                     (t 'false)))))))
(def ∧ fexpr (l)
     (cond ((eq (length l) 2)
            (cond ((and (first l) (second l)) ())
                  (t 'undef)))
           ((is-constant (third l))
                 (cond ((and (first l) (second l))
                        (cond ((or (eq (third l) 'true)
                                   (eq (third l) t)) ())
                              (t 'undef)))
                       ((or (eq (third l) 'undef)
                            (eq (third l) 'false)) ())
                       (t 'undef)))
           (t (list (cons (third l)
                          (cond ((and (first l) (second l)) t)
                                (t 'false)))))))
(def > fexpr (l)
     (cond ((eq (length l) 2)
            (cond ((old> (first l) (second l)) ())
                  (t 'undef)))
           ((is-constant (third l))
                 (cond ((old> (first l) (second l))
                        (cond ((or (eq (third l) 'true)
                                   (eq (third l) t)) ())
                              (t 'undef)))
                       ((or (eq (third l) 'false)
                            (eq (third l) 'undef)) ())
                       (t 'undef)))
           (t (list (cons (third l)
                          (cond ((old> (first l) (second l)) t)
                                (t 'false)))))))
(def = fexpr (l)
     (cond ((eq (length l) 2)
            (cond ((old= (first l) (second l)) ())
                  (t 'undef)))
           ((is-constant (third l))
                 (cond ((old= (first l) (second l))
```

```
                                        (cond ((or (eq (third l) 'true)
                                                   (eq (third l) t)) ())
                                              (t 'undef)))
                                       ((or (eq (third l) 'false)
                                            (eq (third l) 'undef)) ())
                                       (t 'undef)))
                              (t (list (cons (third l)
                                             (cond ((old= (first l) (second l)) t)
                                                   (t 'false)))))))
;             (def + (x y) (+ x y))          these functions are already
;             (def - (x y) (- x y))          defined properly, they are listed
;             (def * (x y) (* x y))          here just to indicate we didn't
;             (def // (x y) (// x y))        forget them
              (def r+ (x y) (+$ x y))
              (def r- (x y) (-$ x y))
              (def r* (x y) (*$ x y))
              (def r// (x y) (//$ x y))

;stuff for adding strings which are represented as a list of 2 elements,
; the first is the atom STRING, the second is a list of the characters in the
; string.
(def readstring ()
     (prog (temp hdr)
           (setq temp (cons (readch) () ))
           (setq hdr (cons temp temp))
           (return (do ((nxtchar (readch) (readch)))

                       ((and (eq nxtchar '")
                             (not (eq (tyipeek) 42)))
                        (list 'string (car hdr)))
                       (cond ((eq nxtchar '")(readch)))
                       (setq temp (cons nxtchar () ))
                       (rplacd (cdr hdr) temp)
                       (rplacd hdr temp))))))

(def string fexpr (l) (cons 'string l))
(setsyntax '" 'macro 'readstring)
(putprop 'prt (get 'print 'subr) 'subr)
(def print (x)
     (cond  ((or (atom x) (not (eq (first x) 'string)))
                          (prt x))
            (t (prt (maknam (second x))))))

(def is-string fexpr (x)
     (cond ((eq (length x) 1)
            ((lambda (y)
```

```
                    (cond
                      ((and (not (atom y))
                            (eq (first y) 'string)
                            (null (rest (rest y)))) ())
                      (t 'undef)))
                  (eval (first x))))
              (t ((lambda (y1 y2)
                    (cond
                      ((and (not (atom y1))
                            (eq (first y1) 'string)
                            (null (rest (rest y1))))
                       (cond ((or (eq y2 'true)
                                  (eq y2 t)) ())
                             ((or (eq y2 'false)
                                  (eq y2 'undef)) 'undef)
                             (t (list (cons y2 t)))))
                      ((or (eq y2 'false)
                           (eq y2 'undef)) ())
                      ((or (eq y2 'true)
                           (eq y2 t)) 'undef)
                      (t (list (cons y2 'false)))))
                  (eval (first x))
                  (second x)))))
(def s-cat (x y)
     (cond ((not (is-string x))
                 (error '(s-cat applied to non-string) x))
           ((not (is-string y))
                 (error '(s-cat applied to non-string) y))
           (t (list 'string (append (second x) (second y))))))
(def firstch (x)
     (cond  ((not (is-string x)) (error '(firstch of non-string)x))
            ((atom (second x)) (error '(firstch of emptystring)x))
            (t (first (second x)))))
(def tail (x)
     (cond  ((not (is-string x)) (error '(tail of non-string) x))
            ((atom (second x)) (error '(tail of emptystring)x))
            (t (list 'string (rest (second x))))))
(def s-cons (x y)
     (cond ((not (eq (flatc x) 1)) (error '(bad character object/, s-cons) x))
           ((not (is-string y)) (error '(s-cons of non-string) y))
           (t (list 'string (cons x (second y))))))
(def mk-string (x) (cond      ((not (eq (flatc x) 1)) (error '(mk-string of non-character)
                                                             x))
                              (t (list 'string (cons x () )))))
(def is-list (l)
     (cond ((atom l) (eq l nil))
```

```
            (t (is-list (rest l)))))

(def firstsym fexpr (l) (matchterms (second l) (list 'quote (ratom))))

(def firstexp fexpr (l) (matchterms (second l) (read)))


(def write (x) (print x) ())



            )

           'primdefs)
NIL) ;end of prog surrounding all the primitive definitions

;the following include all definitions needed for lisp to run generated programs


(def bktrkcond (actuals list-alts)
      (cond ((null list-alts) 'undef)
            (t ((lambda (alt)
                  ((lambda (answer-sub)
                        (cond ((defined answer-sub)
                                    (cleanup answer-sub actuals))
                              (t (bktrkcond actuals (rest list-alts)))))
                   ((lambda (sub)
                         (cond ((defined sub)
                                     (append-if-defined sub
                                          (try (mk-subst (try-pt alt)
                                                         sub))))
                               (t 'undef)))
                    (match actuals (match-pt alt)))))
                (new-version (first list-alts))))))

(def new-version (alt)
      ((lambda (sub)
            (list (mk-subst (match-pt alt) sub)
                  ((lambda (newtry)
                        (cons 'try (mk-subst newtry (chg-formals newtry))))
                   (mk-subst (try-pt alt) sub))))
       (chg-in (match-pt alt)))))


(def chg-formals (sbgoals)
```

```
        (cond ((null sbgoals) ())
              (t ((lambda (x) (cond (x (append x (chg-formals (mk-subst
                                                              (rest sbgoals)
                                                              x))))
                                    (t (chg-formals (rest sbgoals)))))
                  (chg-in (rest (first sbgoals)))))))
(def chg-in (params)
     (cond ((null params) ())
           ((atom (first params))
            (cond
              ((eq (firstchar (first params)) '!)
                 ((lambda (chg) (cons chg (chg-in (subst (cdr chg)
                                                         (car chg)
                                                         (rest params)))))
                  (cons (first params) (gensym))))
                (t (chg-in (rest params)))))
           ((eq (first (first params)) 'quote)
                 (chg-in (rest params)))
           (t ((lambda (chgs)
                 (append chgs (chg-in (mk-subst (rest params) chgs))))
               (chg-in (rest (first params)))))))

(def firstchar (name)
     (first (explode name)))


(def cleanup (sub actuals) (reverse (cleanup* sub actuals () )))

(def cleanup* (sub actuals ans)
     (cond ((null actuals) ans)
           (t ((lambda (act)
                 (cond   ((is-var act)
                          (cond ((already-there act ans)
                                 (cleanup* sub (rest actuals) ans))
                                (t (cleanup* sub (rest actuals)
                                             (cons (cons act
                                                         (lookup* act sub))
                                                   ans)))))
                         ((is-constant act)
                               (cleanup* sub (rest actuals) ans))
                         (t (cleanup* sub (rest actuals)
                                      (cleanup* sub (rest act) ans)))))
               (first actuals)))))


(def already-there (name alist)
```

```
        (cond ((null alist) nil)
              ((eq name (var (first alist))) t)
              (t (already-there name (rest alist)))))

(def lookup* (exp alist)                ;makes all substitutions in alist that are
     (do ((value (mk-subst exp alist) nxtval)                ;applicable to exp
          (nxtval (mk-subst (mk-subst exp alist) alist)
                  (mk-subst nxtval alist)) )
         ((equal value nxtval) value)))


(def try (subgoals)
     (cond ((null subgoals) () )
           (t
                ( (lambda (sub)
                     (cond ((defined sub)
                            (append-if-defined sub
                                               (try (mk-subst
                                                     (rest subgoals)
                                                     sub))))
                           (t 'undef)))
                  (eval (first subgoals)) ))
           ))


(def append-if-defined (sub1 sub2)                    ;when used in conjunction
     (cond ((and (defined sub1) (defined sub2))       ;with "try", this involves
            (append sub1 sub2))                    ; a redundant test on the
           (t 'undef)))                           ;first argument (who cares?)

(def true-precond (fnapp)
     (prog (env2 ans-sub)
           (setq env2 (bind (matchlis (first fnapp)) (rest fnapp)))
           (setq ans-sub (evpred (mk-subst (get (first fnapp) 'precond) env2)))
     (return (cond ((eq ans-sub ()) t)
                   ((eq ans-sub 'undef) nil)
                   (t (error '(variables in precond check) ans-sub))))))


;precondition checker - evpred evaluates wffs
(def evpred (f)
  (cond ((atom f) (cond ((eq f t) ())
                        ((eq f nil) (error '(no precondition exists for)
                                           (first fnapp)))
                        (t (error '(weird atomic predicate)f))))
        ((is-or f) (evor (rest f)))
```

```
        ((is-and f) (evand (rest f)))
        ;otherwise it is a pred-app
        (t (eval f))))


(def is-or (f) (eq (first f) 'v))


(def evor (l) (prog (val)
              (setq val (evpred (first l)))
       (return (cond ((eq 'undef val) (evpred (second l)))
                     ((eq () val) ())
                     (t (evpred (second l))) ;who knows, we might as well let 'em try
                     (t (error '(evpred evaluates to something other than () or undef)
                               (cons val (first l))))))))

(def is-and (f) (eq (first f) '^))


(def evand (l) (prog (val)
              (setq val (evpred (first l)))
       (return (cond ((eq val ()) (evpred (second l)))
                     ((eq val 'undef) 'undef)
                     (t (error '(evpred evaluates to weirdness) (cons val (first l))
                                                                    ))))))


(def defined (x) (not (eq x 'undef)))
;bind is here creating an alist, (a list of "(var.value)" pairs)
(def bind (l1 l2)
       (cond ((null l1) ())
             (t (cons (cons (first l1) (first l2)) (bind (rest l1) (rest l2))))))
(def matchlis (fname) (get fname 'params))


(def match (l1 l2)
    (prog (termatch)
  (return (cond ((null l1) (cond ((null l2) ())
                                 (t (error '(tried to match lists of unequal length)l2
                                                                        ))))
                (t (setq termatch (matchterms (first l1) (first l2)))
                   (prog (restmatch)
                     (return
                       (cond ((null termatch) (match (rest l1) (rest l2)))
```

```
                    ((eq termatch 'undef) 'undef)
                    ((defined
                        (setq restmatch
                            (match (mk-subst (rest l1) termatch)
                                (mk-subst (rest l2) termatch))))
                      (append termatch restmatch))
                    (t 'undef)))) )))))
(def mk-subst (termlist alist)                    ;actually, termlist can be
      (cond ((and (atom termlist)
                  (or (numberp termlist)
                      (eq termlist t)
                      (eq termlist 'f)
                      (eq termlist 'true)
                      (eq termlist 'undef)
                      (eq termlist 'false)
                      (eq termlist nil)))
             termlist)
            ((and (not (atom termlist))
                  (or (not (is-string termlist)) (eq (first termlist) 'quote)))
             termlist)
            (t
             (do ((l alist (rest l))
                  (exp termlist (subst (val (first l)) (var (first l)) exp)))
                 ((null l) exp)))))
(def val (x) (cdr x))
(def var (x) (car x))
;all variables occurring in the terms to be matched are unbound, as substitutions
;are made as we go

(def matchterms (t1 t2)
    (prog (f)
    (return (cond ((equal t1 t2) ())
          ((atom t1) (cond ((is-var t1) (cond ((or (eq(is-string t2)())
                                                   (atom t2)) (cons
                                                                (cons t1 t2)
                                                                () ))
                                              ((occurs-in t1 t2)
                                                  (cond ((get
                                                          (setq f
                                                           (implode (append
                                                             (explode
                                                               'equal-bind-)
                                                             (explode
                                                               (get (first t2)
                                                                'typename)))))
                                                         'body)
```

```
                                        (f t1 t2))
                                       (t 'undef)))
                                (t (cons (cons t1 t2) ())))))
                        (t (cond ((is-var t2) (cons (cons t2 t1) ()))
                                 ((is-constant t1)
                                      (cond ((atom t2) 'undef)
                                            ((and (eq (first t2) 'quote)
                                                  (eq (second t2) t1))
                                             ())
                                            (t 'undef)))
                                 ((is-constant t2) 'undef)
                                 ((contains-var t2) (cond
                                                     ((get
                                                       (setq f
                                                        (implode (append
                                                          (explode
                                                           'equal-bind-)
                                                          (explode
                                                            (get (first t2)
                                                             'typename)))))
                                                       'body)
                                                        (f t1 t2))
                                                       (t 'undef)))

                                 ((get
                                      (setq f (implode (append
                                                        (explode
                                                         'equal-)
                                                        (explode
                                                          (get (first t2)
                                                           'typename)))))
                                      'body)
                                        (cond ((f t1 t2) ())
                                              (t 'undef)))
                                 (t 'undef)))))
((is-var t2) (cond
                    ((eq (is-string t1) ())
                     (cons (cons t2 t1) () ))
                    ((occurs-in t2 t1)
                     (cond ((get
                          (setq f
                           (implode (append
                             (explode
                              'equal-bind-)
                             (explode
                               (get (first t1)
```

```
                                        'typename)))))
                                    'body)
                                     (f t1 t2))
                                    (t 'undef)))
                              (t (cons (cons t2 t1) ())))))
((and (atom t2) (is-constant t2))
      (cond ((and (eq (first t1) 'quote)
                  (eq (second t1) t2))
             ())
            (t 'undef)))
((get (setq f (implode (append (explode 'equal-)
                               (explode (get (first t1)
                                             'typename)))))
   'body)
  (cond ((f t1 t2) ())
        (t 'undef)))
((or (and (not (eq (is-string t1) ()))(contains-var t1))
     (and (not (eq (is-string t2) ()))(contains-var t2)))
      (cond  ((get (setq f (implode (append
                                      (explode
                                        'equal-bind-)
                                      (explode
                                        (get (first t1)
                                          'typename)))))
                   'body)
              (f t1 t2))
             ((and (eq (first t1) (first t2)) (not (eq (first t1) 'quote)))
                (match (rest t1) (rest t2)))
             ((eq (first t1) 'list)
                    (matchterms (cons-out (rest t1)) t2))
             ((eq (first t2) 'list)
                    (matchterms (cons-out (rest t2)) t1))
             ((eq (first t1) 'cons)
              (cond ((eq (first t2) 'quote)
                     ((lambda (x)
                        (cond
                          ((defined x)
                              ((lambda (y)
                                 (cond
                                   ((defined y)
                                      (append x y))
                                   (t 'undef)))
                               (matchterms
                                   (mk-subst (third t1) x)
                                   (list 'quote
                                       (rest (second t2)) ))))
```

```
            (t 'undef)))
          (matchterms (second t1)
                      (list 'quote
                            (first (second t2))))))
         (t
          ((lambda (x)
             (cond
               ((defined x)
                    ((lambda (y)
                       (cond
                         ((defined y)
                           (append x y))
                         (t 'undef)))
                     (matchterms
                         (mk-subst (third t1) x)
                         (mk-subst (rest t2) x))))
               (t 'undef)))
           (matchterms (second t1) (first t2))))))
    ((eq (first t2) 'cons)
     (cond ((eq (first t1) 'quote)
            ((lambda (x)
               (cond
                 ((defined x)
                      ((lambda (y)
                         (cond
                           ((defined y)
                             (append x y))
                           (t 'undef)))
                       (matchterms
                           (mk-subst (third t2) x)
                           (list 'quote
                                 (rest (second t1)) ))))
                 (t 'undef)))
             (matchterms (second t2)
                         (list 'quote
                               (first (second t1))))))
           (t
            ((lambda (x)
               (cond
                 ((defined x)
                      ((lambda (y)
                         (cond
                           ((defined y)
                             (append x y))
                           (t 'undef)))
                       (matchterms
```

```
                                        (mk-subst (third t2) x)
                                        (mk-subst (rest t1) x))))
                                (t 'undef)))
                            (matchterms (second t2) (first t1))))))
                    (t 'undef)))
            (t 'undef)))))


(def cons-out (l)
    (cond ((null l) ())
        (t (list 'cons (first l) (cons-out (rest l))))))


(def contains-var (exp)
    (cond ((atom exp) (is-var exp))
        (t (list-contains-var (rest exp)))))
;contains-var ignores function names when looking for variables since the only
;functions left in at this point are constant functions (arithmetic and constructors)
(def list-contains-var (explist)
    (cond ((null explist) nil)
        ((contains-var (first explist)) t)
        (t (list-contains-var (rest explist)))))
(def occurs-in (var exp)
    (cond ((atom exp) (cond ((eq var exp) t)
                        (t nil)))
        ((occurs-in var (first exp)) t)
        (t (occurs-in var (rest exp)))))
(def match-pt (alternative) (first alternative))
(def try-pt (alternative) (rest (second alternative)))


;*** automatic predicatizing ***
(def autopred (f varlist)
  (cond
    ((get f 'body) nil) ;it's already defined, go away
    (t
      (putprop f (mk-params varlist) 'params)
      (putprop f t 'precond)      ;using system defined functions relys on the
      (putprop f (invent-pat varlist) 'inpat)
      (putprop f t 'postcond)
      (putprop f (mk-predbody f varlist) 'body))))
(def mk-params (varlist)
    (do ((ct varlist (rest ct))
        (l () (cons (intern (gensym)) l)))
      ((null ct) l)))
(def mk-predbody (f varlist)
```

```
(prog (namestring)
(setq namestring (explode f))
(return (cond ((eq (second namestring) 'f)
          (eval (list 'defun f 'fexpr '(argl)
                  (list 'prog '(seplist outpos)
                      '(setq seplist (split argl))
                      '(setq outpos (second seplist))
                    (list 'return
                      (list 'cond
                          (list '(is-var outpos)
                          (list 'list
                          (list 'cons
                          'outpos
                          (list 'eval
                              (list 'cons
                                  (list 'quote
                                      (cond
      ((eq (third namestring) '//)
          (implode (rest (rest (rest namestring)))))
                                              (t
                                              (implode
                                                (rest (rest
                                                    namestring))
                                                ))) )
                                          '(first seplist))) )))
                                  (list (list
                                          'equal
                                          'outpos
                                          (list 'eval
                                              (list 'cons
                                                  (list 'quote
                                                      (cond
      ((eq (third namestring) '//)
          (implode (rest (rest (rest namestring)))))
                                                      (t
                                                      (implode
                                                        (rest (rest
                                                            namestring))
                                                        ))) )
                                                      '(first seplist))) )
                                                  () )
                                  (list t "undef") ))))))
          ((eq (second namestring) 'p)
            (eval (list 'defun f 'fexpr '(argl)
                    (list 'prog '(seplist ans)
                        '(setq seplist (split argl))
```

```
                                (list 'setq
                                      'ans
                                      (list 'eval
                                            (list 'cons
                                                  (list 'quote
                                                        (implode
                                                          (rest (rest
                                                                  namestring))
                                                                ))
                                                  '(first seplist))))
                                '(return
                                       (cond
                                            ((is-var (second seplist))
                                              (list (cons (second seplist)
                                                          ans)))
                                            ((equal ans (second seplist))
                                              () )
                                            (t 'undef)))
                                ))) )
                        (t (error '(autopred called w/no 2p or 2f) f))))))
(def split (l) (cond ((null l) (error '(request to split empty list)l))
                     ((null (rest l)) (list () (first l)))
                     (t ((lambda (h) (cons (cons (first l) (first h))
                                           (rest h)))
                         (split (rest l))))))
(def invent-pat (varlist)
       (do ((ct (rest varlist) (rest ct))
            (pat (cons 0 () ) (cons 1 pat)))
           ((null ct) pat)))
```

16.2 Listing of Pascal Implementation

```
;lisp programs for generating pascal
;each function specification gets turned into a complete program so that
;a library of functions can be built.  Each function is declared external.
;the surrounding program is just a dummy to satisfy syntax restrictions.

;a type specification is turned into a type declaration and stuffed on the
;property list of the type name(under 'type-dec), to be
;included in the declaration of every
;function that uses the type. (apparently there is no such thing as an
;external type in pascal, so it has to be re-declared everywhere)
```

```
(def make_pascal_def ()
        (append
          (list 'program '/ (gensym) '/, name  '/; '/
)
          (make-type-decs) ;makes the definitions for all types used
                    ;i.e., termlists, terms, constants, symbols,
                    ; and whatever else is necessary as subtypes
          (make-external-decs (get name 'external-procs))
          (list 'function '/  name  (rest (make-parameter-list
                                        (strip! (get name 'params))
                                            (get name 'inpat)))
                ': '/  'boolean '/; '/
)
          (make-body-of name);dont forget to include: new(actuals);
          (list 'begin '/                ;       actuals↑.sl := true;
                'end '/.)
        ))


(def strip! (varlist)
     (cond ((null varlist) ())
          (t (cons ((lambda (namelist)
                    (cond ((eq (first namelist) '!)
                           (implode (rest namelist)))
                          (t (first varlist))))
                  (explode (first varlist)))
                 (strip! (rest varlist))))))


(def make-parameter-list (params inpat) ;declares all formals to be type term, but
  (cond ((null params)())              ;leaves an extra ";" on the
        (t (append                     ; front of the list
            ((lambda (arg)
               ((lambda (first-dec)
                  (cond ((eq (first inpat) 0)
                         (append '(/;  var) first-dec))
                        (t (cons  '/; first-dec))))
                (list arg ': 'term)))
             (first params))
            (make-parameter-list (rest params) (rest inpat))))))


;every formal parameter is of type TERM

;every local used by the spec must be of type TERM
```

```
(def make-local-decs (locals)   ;takes a 4 element list of vars to be declared as
      (append '(var /
                                          ;termlists, terms, constants, and
)                                         ;symbols, and returns a list that
        (cons '/    (rest                  ;consists of the pascal to do it
                    (rest (make-decs (first locals)
                                     'termlist))))
        (cons '/    (rest
                    (rest (make-decs (second locals)
                                     'term))))
        (cond ((null (third locals)) ()) (t
        (cons '/    (rest
                    (rest (make-decs (third locals)
                                     'constant))))))
        (cond ((null (fourth locals)) ()) (t
        (cons '/    (rest
                    (rest (make-decs (fourth locals)
                                     'symbol)))))))))

(def make-decs (vars type)      ;makes a list of vars : type, leaves an extra ","
      (cond ((null vars) (list ': '/  type'/;'/
))                                                ;and " " on front
           (t (append (list '/, '/ (first vars))
                      (make-decs (rest vars) type)))))

(def make-type-decs ()
      '(type /
/     alltyps / = / /( integertyp /, realtyp /, booleantyp /,
                     chartyp /, symboltyp /) /; /
/
/     termtyps / =/ /( variable /, / constanttyp /, / funapp /)/;/
/
/     term / =/ †t1 /;/
/
/     termlist/ =/ †tl1/;/
/
/     constant/ =/ †c1/;/
/
/     symbol/ =/ †sym1/;/
/
/     t1/ =/ record/
/     /       case / ttyp:termtyps / of/
/     /       /       variable:/ /(vr:/ integer/)/;/
/     /       /       constanttyp:/ /(cnst:/ constant/)/;/
/     /       /       funapp:/ /(fname:/ symbol/;/
/     /       /       /       / args:/ termlist/)/
```

```
/       /       end/;/
/
/       tl1/ =/ record/
/       /       notempty:/ boolean/;/
/       /       first:/ term/;/
/       /       rest:/ termlist/
/       /       end/;/
/
/       c1/ =/ record/
/       /       case/ ctyp:alltyps/ of/
/       /       /       integertyp:/ /(ival:/ integer/)/;/
/       /       /       realtyp:/ /(rval:/ real/)/;/
/       /       /       booleantyp:/ /(bval:/ boolean/)/;/
/       /       /       chartyp:/ /(cval:/ char/)/;/
/       /       /       symboltyp:/ /(sval:/ symbol/)/
/       /       end/;/
/
/       sym1/ =/ record/
/       /       notempty:/ boolean/;/
/       /       firstch:/ char/;/
/       /       tail:/ symbol/;/
/       /       end/;/
/
/       varpairs/ =/ fvp/;/
/
/       vp/ =/ record/
/       /       notempty:/ boolean/;/
/       /       old:/ integer/;/
/       /       new:/ integer/;/
/       /       rest:/ varpairs/
/       /       end/;/
/
))


(def cons-if-new (x l)
        (cond ((memq x l) l)
              (t (cons x l))))


(def general-funs ()
        '(function / occur(x/,/ y:/ term):/ boolean/;/
/       extern/;/
        function/ genvar:/ integer/;/
/       extern/;/
        procedure / replace(x/,/ t:/ term/;/ var/ tml:/ termlist)/;/
```

```
/       extern/;/
        procedure/ subst(x/,/ t:/ term/;/ var/ t1/,/ t2:/ termlist)/;/
/       extern/;/
        function/ eqsym(x/,/ y:/ symbol):/ boolean/;/
/       extern/;/
        function/ eqconst(x/,/ y:/ constant):/ boolean/;/
/       extern/;/
        function/ copysym(oldsym:/ symbol):/ symbol/;/
/       extern/;/
        function/ copyterm(oldtm:/ term):/ term/;/
/       extern/;/
        function/ copytermlist(tml:/ termlist):/ termlist/;/
/       extern/;/
        function/ copyconst(oldconst:/ constant):/ constant/;/
/       extern/;/
        function/ unify(var/ x/,y/,allx/,ally:termlist/;failed:boolean):/ boolean/;/
/       extern/;/
        procedure/  Lookup(tm:/ term/;/ tbl:/ varpairs/;/ found:/ boolean)/;/
/       extern/;/
        procedure/ Standapart(tml:/ termlist/;/ var/ donetbl:/ varpairs)/;/
/       extern/;/
))



(def make-external-decs (procnames)
      (cond ((null procnames) (general-funs))
            (t (append (mk-ext-dec (first procnames))
                       (make-external-decs (rest procnames)))))))


(def mk-ext-dec (name)
      ((lambda (x)
            (cond (x x)
                  (t (putprop name (list 'FUNCTION '/  name
                                         (make-parameter-list (get name 'types)
                                                              (get name 'inpat))
                                         ': '/ 'boolean '/; '/
'/                                       'EXTERN '/; '/
)
                                       'extproc-head))))
       (get name 'extproc-head)))


(def algol-ize (wff)
      (cond ((atom wff) 'true)
```

```
((eq (first wff) '∧)
 ((lambda (arg1 arg2)
       (cond ((eq arg1 'true) (cond ((eq arg2 'true) 'TRUE)
                                    (t arg2)))
             ((eq arg2 'true) arg1)
             (t (list arg1 'AND arg2))))
  (algol-ize (second wff))
  (algol-ize (third wff))))
((eq (first wff) '∨)
 ((lambda (arg1 arg2)
       (cond ((eq arg1 'true) 'TRUE)
             ((eq arg2 'true) 'TRUE)
             (t (list arg1 'OR arg2))))
  (algol-ize (second wff))
  (algol-ize (third wff))))
(t  ((lambda (ans-code)          ;o.w. it is a funapp, so generate
       (setq pascode (append pascode     ;pascal to represent the
                             (rest ans-code))) ;terms and build the
       (list (first wff)                       ;new call
             (first ans-code)))
     (actualize (rest wff))))))
```

```
(def make-try (trylist)              ;gets called with the list of subgoals with
     (cond ((null trylist) 'true)    ;the "try" stripped off, and generates a
           (t ((lambda (call)        ;conjunction out of the subgoal calls
                 (cond ((null (rest trylist))
                        call)
                       (t (append call
                                  '(AND)
                                  (make-try (rest trylist))))))
               (list (first (first trylist))
                     (rest (first trylist)))))))
```

```
(def stripdeep! (exp)
  (cond ((atom exp) ((lambda (namelist)
                       (cond ((eq (first namelist) '!)
                              (implode (rest namelist)))
                             (t exp)))
                     (explode exp)))
        (t (cons (stripdeep! (first exp))
                 (stripdeep! (rest exp))))))
```

```
(def make-body-of (name)
  (prog (donelist local-vars pascode)
    (setq local-vars '((actuals copyactuals matchlist) () () ()))
    (setq donelist (strip! (get name 'params)))
    (setq pascode
     (append
      '(BEGIN /
)     pascode
      '( if /
)
      (list (algol-ize (stripdeep! (get name 'typedprecond))) '/
)
      '(then / begin /
)
      (build-actuals (strip! (get name 'params)))
      (do ((alts (reverse (stripdeep! (rest (get name 'body))))) (rest alts))
          (ans () ((lambda (alt)
                   (setq donelist '(actuals copyactuals matchlist))
                   (append
                    '(copyactuals / := / copytermlist (actuals) /; /
                      new(donetbl)/;/
                      donetbl↑/.notempty/ :=/ false/;/
                      standapart (copyactuals/, donetbl)/;/
)
                    (mk-termlist 'matchlist (match-pt alt))
                    '(if /  unify (copyactuals /, matchlist/, copyactuals/,
                                   matchlist/, failed) /
                      then / begin /
)
                    ((lambda (sbgls-code)
                       (append (rest sbgls-code)
                               (list 'failed '/ ':= '/  'not '/
                                     (make-try (first sbgls-code)) '/
                                     'end '/
)))
                     (fix-subgoals (try-pt alt)))
                    '(else / failed / := / true /; /
)
                    ans))
                  (first alts))))
         ((null alts) ans))
      (list 'flag '/ ':= '/ 'not '/ 'failed '/; '/
            name '/ ':='/ 'flag'/;'/
            'if '/  'flag '/
            'then '/ 'begin '/
)
```

```
        (mk-assgns (strip! (get name 'params)))
        (list 'end '/
              'end '/
              'else '/  name '/ ':= '/  'false '/
              'end '/;'/
)))
   (return
        (append
          (make-local-decs local-vars)
          '(/  donetbl:/ varpairs/;/
/              flag/,/ failed:/ boolean/;/
/
)         pascode))))



(def mk-assgns (params)
        (cond ((null params) ())
              (t (append (list (first params) '/ ':= 'copyactuals↑/.first '/; '/
                               'copyactuals '/ ':='/ 'copyactuals↑/.rest '/; '/
)                       (mk-assgns (rest params))))))


;build-actuals generates the pascal code to build a termlist out of the actual
;parameters, the actuals are already of type term, so all it has to do is link
;them together into a termlist called actuals so they will be appropriate input
;to the unifier.
(def build-actuals (params)
   (append
        (list 'new '(actuals) '/; '/
              'actuals↑/.notempty '/ ':= '/ 'false '/; '/
)
        (do ((vars params (rest vars))
             (ans () ((lambda (temp)
                        (addlocal-tml temp)
                        (append
                          (list 'new '/( temp '/) '/; '/
                                temp '↑'/.'notempty '/ ':= '/ 'true'/; '/
                                temp '↑ '/.'first '/ ':= '/ (first vars) '/;'/
                                temp '↑'/.'rest '/ ':= '/ 'actuals'/; '/
                               'actuals '/ ':= '/  temp '/; '/
)
                        ans))
                      (gensym))))
            ((null vars) ans))))
```

```
(def mk-termlist (name arglist) ;generates the pascal code to make a termlist,
      (addlocal-tml name)     ;pointed to by name, whose elements are made up
                              ;of terms constructed from the elements of arglist.
                              ;donelist is global (local to make-body) and is
                              ;re-initialized whenever a new alternative is
                              ;being translated.  All variables created in this
                              ;process (new'd) must be added to the list local-
                              ;vars so that the appropriate declarations will be
                              ;generated for them.
  (append
    (list 'new '/( name '/) '/; '/
          name '↑ '/. 'notempty '/ ':=/ 'false '/; '/
)
    (do ((args (reverse arglist) (rest args))
         (ans () ((lambda (temp)
                    (addlocal-tml temp)
                    (append
                      (list 'new '/( temp '/) '/; '/
                            temp '↑ '/. 'notempty '/ ':=/ 'true '/;'/
)
                      (cond ((is-var (first args))
                             (cond ((memq (first args) donelist)
                                    (list temp '↑'/.'first'/ ':='/
                                          (first args)'/;'/
))
                                   (t (mark-done (first args))
                                      (append (mk-term (first args)
                                                       (first args))
                                              (list temp '↑'/.'first '/ ':='/
                                                    (first args) '/; '/
)))))
                            (t ((lambda (tmname)
                                  (append (mk-term tmname (first args))
                                          (list temp'↑'/.'first '/ ':='/
                                                tmname '/; '/
)))
                                (gensym))))
                      (list temp'↑'/.'rest '/ ':= '/  name '/; '/
                            name '/ ':='/  temp '/; '/
)
                      ans))
                  (gensym))))
         ((null args) ans))))
```

```
(def mk-term (name arg)
  (prog (quotflag)
    (addlocal-tm name)
    (return
      (append
        (list 'new '/( name '/) '/; '/
)
        (cond ((is-var arg)
                (list name'↑'/.'ttyp '/ ':= '/ 'variable'/; '/
                     name '↑'/.'vr '/ ':= '/ 'genvar  '/; '/
))
              ((is-constant arg)
                (cond ((is-quoted arg) (setq quotflag t)
                                       (setq arg (second arg))))
                (cond ((atom arg)
                       (append (list name '↑'/.'ttyp '/ ':='/ 'constanttyp'/; '/
)                              ((lambda (con)
                                 (append
                                  (mk-const con arg)
                                  (list name '↑'/.'cnst '/ ':='/  con'/;'/
)))
                                 (gensym)))))
                      (quotflag
                        (append (list name'↑'/.'ttyp '/ ':='/ 'funapp'/; '/
)                               (mk-sym 'cons (explode 'cons))
                                ((lambda (tml)
                                   (append
                                    (mk-termlist tml (rest (cons-out arg)))
                                    (list name'↑'/.'fname'/ ':='/ 'cons '/;'/
                                     name'↑'/.'args '/ ':='/  tml'/; '/
)))
                                 (gensym)))))
                      (t (append
                           (list name'↑'/.'ttyp '/ ':='/ 'funapp'/; '/
)                          (mk-sym (first arg) (explode (first arg)))
                           ((lambda (tml)
                              (mk-termlist tml (rest arg)))
                            (gensym))
                           (list name'↑'/.'fname'/ ':='/ (first arg)'/;'/
                                name'↑'/.'args '/ ':='/  tml'/; '/
)))))
              (t (append
                   (list name'↑'/.'ttyp '/ ':='/ 'funapp'/; '/
)                  (mk-sym (first arg) (explode (first arg)))
                   ((lambda (tml)
                      (mk-termlist tml (rest arg)))
```

```
                    (gensym))
                   (list name'↑'/.'fname'/ ':='/  (first arg)'/;'/
                        name'↑'/.'args '/ ':='/  tml'/; '/
)))))))


(def cons-out (list)
      (cond ((null list) ())
            (t (list 'cons (cond ((is-constant (first list)) (first list))
                                 (t (list 'quote (first list))))
                           (cons-out (rest list))))))


(def mk-const (name atm)
      (addlocal-cnst name)
      (append (list 'new '/( name '/) '/;'/
)
            (cond ((fixp atm) (list name'↑'/.'ctyp '/ ':= '/ 'integertyp'/; '/
                                    name'↑'/.'ival '/ ':= '/  atm '/; '/
))
                  ((floatp atm) (list name'↑'/.'ctyp '/ ':='/ 'real'/; '/
                                    name'↑'/.'rval '/ ':='/  atm '/; '/
))
                  ((eq atm t) (list name'↑'/.'ctyp '/ ':='/ 'boolean'/; '/
                                    name'↑'/.'bval '/ ':= '/ 'true'/; '/
))
                  ((eq atm false) (list name'↑'/.'ctyp '/ ':='/ 'boolean'/; '/
                                       name'↑'/.'bval '/ ':= '/ 'false'/; '/
))
                  (t (append (list name'↑'/.'ctyp '/ ':='/ 'symbol'/; '/
)                          (mk-sym arg (explode arg))
                           (list name'↑'/.'sval '/ ':= '/  arg'/; '/
))))))


(def is-quoted (x) (cond ((atom x) nil)
                         (t (eq (first x) 'quote))))


(def mk-sym (name charlist)
  (cond
    ((memq name donelist) ())
    (t
      (addlocal-sym name)
      (append (list 'new '/( name '/)'/;'/
```

```
                name '↑ '/. 'notempty '/ ':='/ 'false'/;'/
)
            (do ((chars (reverse charlist) (rest chars))
                (ans () ((lambda (temp)
                            (addlocal-sym temp)
                            (append
                              (list 'new '/( temp '/)'/; '/
                                    temp '↑'/.'notempty '/ ':='/ 'true'/;'/
                                    temp '↑'/.'firstch '/ ':='/ (first chars)'/;'/
                                    temp '↑'/.'tail '/ ':='/  name'/; '/
                                    name '/ ':='/  temp '/;'/
)
                            ans))
                         (gensym)))))
              ((null chars) ans))))))


(def fix-subgoals (sbglist)     ;makes a list of subgoals whose arglists are lists
  (cond                         ;of terms for which the pascal code has been
    ((null sbglist)'(()))       ;generated. its value is the new sbglist cons onto
    (t ((lambda (first-ans rest-ans)  ;the list of pascal stuff
        (cons
         (cons (cons (first (first sbglist)) ;i.e., subgoal name
                     (first first-ans))      ;i.e., new arglist
               (first rest-ans))             ;the other subgoals
         (append (rest first-ans)            ;p-code for this subgoal
                 (rest rest-ans))))          ;p-code for the rest of the subgoals
        (actualize (rest (first sbglist)))   ;generates stuff for one arglist
        (fix-subgoals (rest sbglist))))))


(def actualize (arglist)        ;turns an arglist in intermediate form, into a list
  (cond                         ;of terms- the value is the new arglist cons'd onto
    ((null arglist)'(()))       ;the list of p-code
    (t ((lambda (arg-ans rest-ans)
          (cons (cons (first arg-ans)
                      (first rest-ans))
                (append (rest arg-ans)
                        (rest rest-ans))))
        (arg-to-term (first arglist))
        (actualize (rest arglist))))))


(def arg-to-term (arg)
  (cond
    ((is-var arg)
```

```
        (cond ((memq arg donelist) (list arg))
              (t (cons arg (mk-term arg arg)))))
      (t ((lambda (name)
            (cons name
                  (mk-term name arg)))
          (gensym)))))


;local-vars and donelist are global to several of the above functions. They are
;the means by which information about which terms have been generated and
;must be declared are transmitted about. Local-vars is a list of 4 lists of
;variables that must be declared as termlists, terms, constants, and symbols,
;respectively. Every time a new variable is created it is added to this list
;in the appropriate type sublist. Whenever a variable (pointer) is created
;that may appear again, it is added to donelist so that multiple versions
;need not be generated.


(def mark-done (name) (setq donelist (cons-if-new name donelist)))

(def addlocal-tml (name)
  (setq local-vars (cons (cons-if-new name (first local-vars))
                         (rest local-vars))))

(def addlocal-tm (name)
  (setq local-vars (cons (first local-vars)
                         (cons (cons-if-new name (second local-vars))
                               (rest (rest local-vars))))))

(def addlocal-cnst (name)
  (setq local-vars (cons (first local-vars)
                         (cons (second local-vars)
                               (cons (cons-if-new name (third local-vars))
                                     (rest (rest (rest local-vars)))))))))

(def addlocal-sym (name)
  (setq local-vars (cons (first local-vars)
                         (cons (second local-vars)
                               (cons (third local-vars)
                                     (cons (cons-if-new name (fourth local-vars))
                                           ()))))))
```

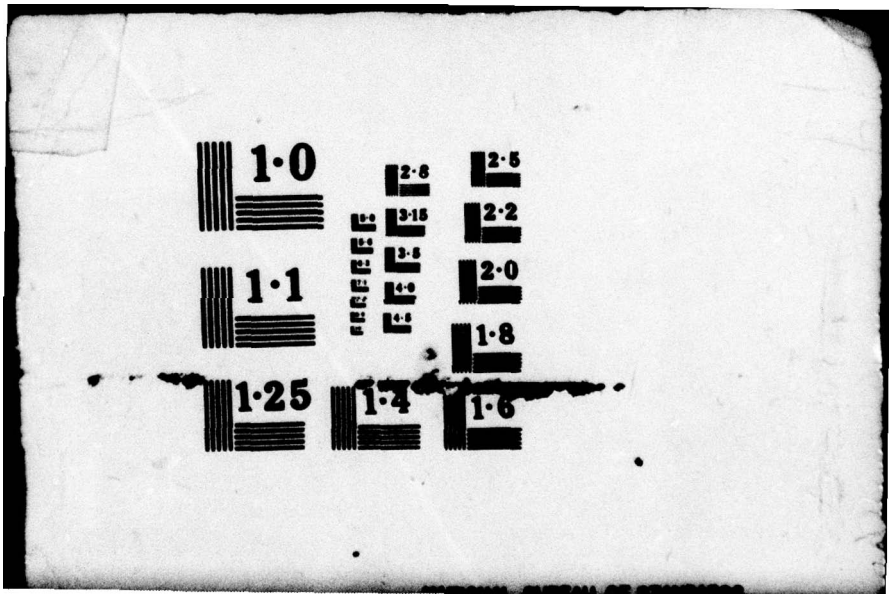
The following programs were written directly in Pascal as part of the implementation of the "back end" for the language.

```
(*pascal programs*)
(*$E+*)
program Junk,Unify,Subst,Replace,Copytermlist,Copyterm,Copyconst,Copysym;

(*pascal can't handle things the way it should so we have to invent strange
      names that are all referring to the same thing, in particular, the type
      of the object at hand. Thus,
      alltyps, an indication of the possible atomic types, is actually made
            up of convoluted versions of the type names.
      this idiocy is carried on throughout, which is why you'll see several
      different names that all look similar but had to be different for pascal. *)

TYPE
      (* Alltyps are the types of atomic constants *)
      Alltyps = (Integertyp, Realtyp, Booleantyp, Chartyp, Symboltyp);

      Termtyps = (Variable, Constanttyp, Funapp);

      Term = ↑T1;

      Termlist = ↑T11;

      Constant = ↑C1;

      Symbol = ↑Sym1;

      T1 = record
            case Ttyp:Termtyps of
                  Variable: (Vr: integer);
                  Constanttyp: (Cnst: Constant);
                  Funapp:   (Fname: Symbol;
                              Args: Termlist)
            end;

      T11 = record
            Notempty: Boolean;
            First: Term;
            Rest: Termlist
            end;


      C1 = record
```

AD-A073 023 CALIFORNIA UNIV SANTA CRUZ INFORMATION SCIENCES F/G 9/2
GENERATING CORRECT PROGRAMS FROM LOGIC SPECIFICATIONS.(U)
MAY 79 R E DAVIS N00014-76-C-0682
UNCLASSIFIED TR-79-5-001 NL

3 OF 3
AD
A073023

END
DATE
FILMED
9 79
DDC


1·0
1·1
1·25
1·4
1·6
1·8
2·0
2·2
2·5
2·8
3·15
3·5
4·0
4·5

```
            case Ctyp:Alltyps of
                  Integertyp: (Ival: integer);
                  Realtyp:    (Rval: real);
                  Booleantyp: (Bval: boolean);
                  Chartyp:    (Cval: char);
                  Symboltyp:  (Sval: Symbol)
            end;

      Sym1 = record
            Notempty: boolean;
            Firstch: char;
            Tail: Symbol
            end;

      Varpairs = ↑Vp;

      Vp = record
            Notempty: boolean;
            Old: integer;
            New: integer;
            Rest: Varpairs
            end;


function Genvar:integer;
begin
  Genvar:= realtime
end;(*genvar*)

function Occur(X,Y:Term):boolean;
var Ptr: Termlist;
   Flag: boolean;
begin
  if Y↑.Ttyp = Variable
    then begin
          if Y↑.Vr = X↑.Vr
            then Occur := true
            else Occur := false
         end
    else if Y↑.Ttyp = Constanttyp
          then Occur := false
          else begin
                Ptr := Y↑.Args;
                Flag := false;
                while Ptr↑.Notempty and (Flag = false)
                do begin
```

```
                    Flag := Occur(X, Ptr↑.First);
                    Ptr := Ptr↑.Rest
                    end;
                  Occur := Flag
                end
end;(*Occur*)


procedure Replace(X, T: Term; var Tml: Termlist);
  var T11:Termlist;
      T1:Term;
  begin
    T11:= Tml;
    while T11↑.Notempty do
    begin
      T1 := T11↑.First;
      if not(T1↑.Ttyp = Constanttyp)
        then begin
             if T1↑.Ttyp = Variable
             then begin
                  if X↑.Vr = T1↑.Vr
                    then
                        T11↑.First := T (*need to mung record, not just ptr t1*)
                    (*else, no change needed*)
                  end
             else (*it's a funapp*)
                  Replace(X, T, T11↑.First↑.Args)
          end;
      (*if its a constant no changes need be made*)
      T11 := Tml↑.Rest
    end (*of while*)
  end; (*Replace*)

procedure Subst(X, T:Term; var T1, T2:Termlist);
begin
  Replace(X, T, T1);
  Replace(X, T, T2)
end;


function Eqsym(X,Y:Symbol):boolean;
begin
  while X↑.Notempty and Y↑.Notempty and (X↑.Firstch = Y↑.Firstch) do
    begin
      X:=X↑.Tail;
      Y:=Y↑.Tail
```

```
    end;
  if X↑.Notempty or Y↑.Notempty
    then Eqsym:= false
    else Eqsym:= true
end;


function Eqconst(X,Y:Constant):boolean;
begin
  if X↑.Ctyp = Y↑.Ctyp
    then case X↑.Ctyp of
              Integertyp: Eqconst:= X↑.Ival = Y↑.Ival;
              Realtyp: Eqconst:= X↑.Rval = Y↑.Rval;
              Booleantyp: Eqconst:= X↑.Bval = Y↑.Bval;
              Chartyp: Eqconst:= X↑.Cval = Y↑.Cval;
              Symboltyp: Eqconst:= Eqsym(X↑.Sval, Y↑.Sval)
         end
    else Eqconst:= false
end;


function Copysym(Oldsym:Symbol):Symbol;
  var Newsym, Lastnode, Newnode:Symbol;

  begin
    new(Newsym);
    Lastnode := Newsym;
    while Oldsym↑.Notempty do
    begin
      Lastnode↑.Notempty := true;
      Lastnode↑.Firstch := Oldsym↑.Firstch;
      new(Newnode);
      Lastnode↑.Tail := Newnode;
      Lastnode := Newnode;
      Oldsym := Oldsym↑.Tail
    end;
    Lastnode↑.Notempty := false;
    Copysym := Newsym
  end; (*Copysym*)


function Copyterm(Oldtm:Term):Term;
forward;


function Copytermlist(Tml:Termlist):Termlist;
```

```
 var Newnode, Lastnode, Tminew:Termlist;
 begin
  new(Tminew);
  Lastnode := Tminew;
  while Tml↑.Notempty do
  begin
   Lastnode↑.Notempty := true;
   Lastnode↑.First := Copyterm(Tml↑.First);
   new(Newnode);
   Lastnode↑.Rest := Newnode;
   Lastnode := Newnode;
   Tml := Tml↑.Rest;
  end;
  Lastnode↑.Notempty := false;
  Copytermlist := Tminew
 end; (*Copytermlist*)


function Copyconst(Oldconst:Constant):Constant;
 var Newconst:Constant;

 begin
  new(Newconst);
  Newconst↑.Ctyp := Oldconst↑.Ctyp;
  case Newconst↑.Ctyp of
      Integertyp: Newconst↑.Ival := Oldconst↑.Ival;
      Realtyp: Newconst↑.Rval := Oldconst↑.Rval;
      Booleantyp: Newconst↑.Bval := Oldconst↑.Bval;
      Chartyp: Newconst↑.Cval := Oldconst↑.Cval;
      Symboltyp: Newconst↑.Sval := Copysym(Oldconst↑.Sval)
  end; (*of case stmt*)
  Copyconst := Newconst
 end; (*Copyconst*)


function Copyterm;
 var Newtm:Term;

 begin
  new(Newtm);
  Newtm↑.Ttyp := Oldtm↑.Ttyp;
  case Newtm↑.Ttyp of
      Variable: Newtm↑.Vr := Oldtm↑.Vr; (*it's just an integer*)
      Constanttyp: Newtm↑.Cnst := Copyconst(Oldtm↑.Cnst);
      Funapp:  begin
                 Newtm↑.Fname := Copysym(Oldtm↑.Fname);
```

```
                Newtm↑.Args := Copytermlist(Oldtm↑.Args)
              end
  end; (*of case stmt*)
  Copyterm := Newtm
end; (*Copyterm*)


(*the first call on unify will repeat the arglists being unified- dumb, but it
makes it possible to accomplish the substitutions by replacement as we go instead
of building a substitution and making new copies of everything every time we do
a substitution.  the allx and ally args are necessary to ensure that any replacements
resulting from recursive calls get made throughout the entire termlists you started
with*)
function Unify(var X, Y, Allx, Ally: Termlist; Failed:boolean): boolean;
 var X1, Y1: Termlist;
    X2, Y2: Term;
    Dummy, Subfailed: boolean;
 begin
     (*initialize*)
  Failed := False;
  X1 := X;
  Y1 := Y;
  while X1↑.Notempty and Y1↑.Notempty and not(Failed) do
   begin
    X2 := X1↑.First;
    Y2 := Y1↑.First;
    if X2↑.Ttyp = Variable
     then begin
          if Y2↑.Ttyp = Variable
           then X2↑.Vr := Y2↑.Vr
            (* if they're already the same, the assignment is unnecessary
             but cheaper than testing the equality and won't hurt anything*)
           else if Occur(X2, Y2)
                 then Failed := true
                 else Subst(X2, Y2, X, Y)
         end
     else if Y2↑.Ttyp = Variable
          then begin
                if Occur(Y2, X2)
                 then Failed := true
                 else Subst(Y2, X2, X, Y)
               end
          else if X2↑.Ttyp = Constanttyp
                then begin
                      if Y2↑.Ttyp = Constanttyp
                       then begin
```

```
                                    if not( Eqconst(X2↑.Cnst, Y2↑.Cnst) )
                                      then Failed := true;
                                      (*if they are = nothing need be done*)
                                    end
                                  else Failed := true
                                end
                              else (*X2 is a funapp and Y2 is not a variable*)
                                if Y2↑.Ttyp = Constanttyp
                                  then Failed := true
                                  else (*X2 and Y2 are both funapp terms*)
                                    if Eqsym(X2↑.Fname, Y2↑.Fname)
                                      then begin
                                             Dummy :=
                                             Unify(X2↑.Args, Y2↑.Args,
                                                   Allx, Ally, Subfailed);
                                             if Subfailed
                                               then Failed := true
                                           end
                                      else Failed :=true;
        X1 := X1↑.Rest;
        Y1 := Y1↑.Rest
      end; (*of while*)
    if X1↑.Notempty or Y1↑.Notempty then Failed := true;
    Unify := not Failed
  end; (*Unify*)

function Lessthan(X, Y:Term; var Z:Term):boolean;
var Con: Constant;
begin
  Z↑.Ttyp := Constanttyp;
  new(Con);
  Z↑.Cnst := Con;
  Con↑.Ctyp := Booleantyp;
  if X↑.Cnst↑.Ival < Y↑.Cnst↑.Ival
    then Z↑.Cnst↑.Bval := true
    else Z↑.Cnst↑.Bval := false;
  Lessthan := true
end;

function Greaterequal(X, Y:Term; var Z: Term): boolean;
var Con: Constant;
begin
  Z↑.Ttyp := Constanttyp;
  new(Con);
  Con↑.Ctyp := Booleantyp;
  Z↑.Cnst := Con;
```

```
  if X↑.Cnst↑.Ival >= Y↑.Cnst↑.Ival
    then Z↑.Cnst↑.Bval := true
    else Z↑.Cnst↑.Bval := false;
  Greaterequal := true
end;

function Times(X, Y:Term; var Z:Term): boolean;
var Con:Constant;
begin
  Z↑.Ttyp := Constanttyp;
  new(Con);
  Con↑.Ctyp := Integertyp;
  Z↑.Cnst := Con;
  Con↑.Ival := X↑.Cnst↑.Ival * Y↑.Cnst↑.Ival;
  Times := true
end;

function Sub1(X: Term; var Y:Term):boolean;
var Con:Constant;
begin
  Y↑.Ttyp := Constanttyp;
  new(Con);
  Con↑.Ctyp := Integertyp;
  Y↑.Cnst := Con;
  Con↑.Ival := X↑.Cnst↑.Ival - 1;
  Sub1 := true
end;


procedure Lookup(Tm:Term; Tbl: Varpairs; Found: boolean);
var Ptr: Varpairs;
begin
  Found := false;
  Ptr := Tbl;
  while Ptr↑.Notempty and not Found
  do begin
       if Ptr↑.Old = Tm↑.Vr
         then begin
              Tm↑.Vr := Ptr↑.New;
              Found := true
             end;
       Ptr := Ptr↑.Rest
     end
end; (*Lookup*)
```

```
procedure Standapart(Tml: Termlist; var Donetbl: Varpairs);
var Ptr: Termlist;
    Done: Varpairs;
    Found: boolean;
begin
  Ptr:= Tml;
  while Ptr↑.Notempty
  do begin
       if Ptr↑.First↑.Ttyp = Variable
        then begin
             Lookup(Ptr↑.First, Donetbl, Found);
             if not Found
              then begin
                   new(Done);
                   Done↑.Notempty := true;
                   Done↑.Old := Ptr↑.First↑.Vr;
                   Done↑.New := Genvar;
                   Ptr↑.First↑.Vr := Done↑.New;
                   Done↑.Rest := Donetbl;
                   Donetbl := Done
                  end
            end
        else if Ptr↑.First↑.Ttyp = Funapp
              then Standapart(Ptr↑.First↑.Args, Donetbl);
       Ptr := Ptr↑.Rest
    end
end; (*Standapart*)


begin (*Junk, ie, main program*)
end.
```

OFFICIAL DISTRIBUTION LIST

Contract N00014-76-C-0682

Defense Documentation Center
Cameron Station
Alexandria, VA 22314
12 Copies

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 Copies

Office of Naval Research
Code 200
Arlington, VA 22217
1 Copy

Office of Naval Research
Code 458
Arlington VA, 22217
1 Copy

Office of Naval Research
Branch Office, Boston
Bldg. 114, Section D
666 Summer Street
Boston, MA 02210
1 Copy

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, ILL 60605
1 Copy

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
1 Copy

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 Copies

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps (Code RD-1)
Washington, D.C. 20380
1 Copy

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
1 Copy

Mr. E.H. Gleissner
Naval Ship Research & Development Center
Computation and Mathematics Department
Bethesda, MD 20084
1 Copy

Captain Grace M. Hopper (008)
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
1 Copy

Mr. Glick (R53)
Director
National Security Agency
Fort G.G.
Meade, MD 20755
1 Copy