A PROLOG PROTOTYPE OF A MODULE DEVELOPMENT SYSTEM

by

Marita E. Peak

B.S., Kansas State University

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

William J. Hankley
Major Professor

Table of Contents

Chapter Page

List of Figures

CHAPTER ONE : PROTOTYPING OF A MODULE DEVELOPMENT SYSTEM

1.1 INTRODUCTION

There is a critical need for the software industry to ensure the quality of development software and at the same time, increase the productivity of programmers. The most promising solution to these software engineering needs is automation to aid the process of creating and maintaining software. The challenge is to determine the best way to apply automation to software engineering tasks.

Present software engineering methods are not meeting the needs of developers now[Mus85], [Fre85]. These traditional methods require software designers to develop a complete description of the specifications as a first step and then use 'top-down' design methods to subdivide the system into smaller and smaller modules. This method aids in partitioning the complexity of the system. Once the module design is completed, programmers can begin the arduous task of implementing the system. There is little recourse if the specifications were not described correctly other than to start over. There is also a lack of facilities for reusing previous designs or sections of program code. Though the traditional methods are inadequate, alternatives

must be found that are more flexible and manage the complexity of software design.

Researchers seeking to apply automation to software engineering are exploring two promising solutions to aid in the creation of software systems. One solution is the use of prototypes of software systems, and the other is automated program generation using reuseable libraries of code components [Yeh83].

Rapid prototyping is a method of increasing dialogue between the end-users of a software system, and the designers and implementors. A rapid prototype is an executable model of the intended system that shows how the final system will function. Prototypes let users know if the initial specification is inadequate, and also provides insight into interactions of the system that were not explicit when the system was specified. If the user wants changes or additions, they are easy to incorporate. When everyone is satisfied, then the actual system can be written.

The prototyping appoach approach is novel for software systems. Engineers developing industrial products prototype their products to insure that the final system will indeed satisfy customers, and to assist in pinpointing prohlems in design. Software was never thought of as a product that was amenahle to prototying. The reason for the change in attitude is

the recent development of tools capable of directly executing system specifications.

The second solution is to automatically generate programs from reuseable libraries of code components. These components need to be expressed as abstract algorithms which can be capable of being used flexibly, on a variety of different data types that cannot be known until a particular program is being designed. The algorithms thus must be parameterized on all component bindings. Ideally the algorithms could be expressed separately from the syntax of any particular programming language. Maintenance of systems designed in this way would be much easier. The code would be virtually bug free, due to previous use. In order to structure the knowledge of code into pieces that can be integrated into many different instances of programs, organization of the fragments must occur so that they can be retrieved when needed. New fragments must be easily integrated into the system as programmers increase their repertoire of algorithms.

A software environment containing a number of tools with knowledge about the construction, development and maintenance is the key to future software productivity. A high-level language that helps design non-procedural system specifications ; and transformational systems that can use these specifications and

synthesize correct efficient software systems are two key components of this future software environment. The transformation system will need a knowledge-base of program fragments to use in it's task of constructing software systems.

There are two basic divisions in experimental systems that researchers have been working on. The divisions are between general purpose systems, that are trying to embed human-like intelligence in the knowledge and inference of programs, and target domain systems, that do not attempt to make that claim, but aim at developing more immediately useful, and powerful systems. Research into both areas is important.

Currently users are tired of the experimentation and want some immediately useful tools to assist in software development and support. As a researcher, the question that must be addressed is the targeted domain of your system. Is your system an immediately useful tool, or is it theoretically promising, but for now limited?

1.2 SCOPE OF THESIS

In this thesis a number of representative program transformation system will be examined, and they will be classified according to the general-purpose versus domain-specific approach, as well as according to

methods used for program specification, and transformation methodology. Often the systems have overlapping features. Since the problem of developing automated tools in the area of software engineering is so diverse, with researchers attacking from many different angles, it is often difficult to say which is the 'best' approach. Criteria must include the user interface, the scope and efficiency of useable programs that are produced, the correctness and reliability of the system itself, and the amount of computer and human resources the system requires to run properly. [Bal73]

Much research has been done on general purpose transformation systems. The theory of reasoning about the programming process, and knowledge representation is thus being developed. Deductive systems such as Manna and Waldinger's transformation system DEDALUS, and Barstow's PECOS are powerful systems. However they are viewed as research tools, and the programs they synthesize are targeted at the language LISP. The class of programs they work successfully on are 'toys'. Attempts are now being made to extend both the power and knowledge of these systems, and target other languages for development; however the systems are still experimental and 'real' applications can't be counted on from systems such as these for quite a while.

Application generation systems in use now in production environments fit into the target-specific transformation systems. The most successful are systems that generate applications for using databases [Hor85]. For example, users specify particular display screens, by stating where he wants fields, and what values he wants returned to him, and the system generates a COBOL or PL/I program with database calls. Report generation is done in the same manner. If the target programs have a basically similar functionality, they just need to be changed in details, then either a very high level language that 'chunks' the basic functionality (APL with matrix manipulation, Prolog with parsing, database fourth generation languages) can be developed, or a system that can take a basic algorithm from a store, and parameterize it can be developed to automate the production of such programs. The classification task for these metaprogramming systems is made all the more difficult by their similarities. Different approaches are used, but all systems have four basic components they address, that of the user interface, that of storing or processing programming knowledge, that of deciding how to apply this knowledge (inference approach), and that of producing specific target programs. Therefore the major differences have to do with different forms of knowledge representation, and inference [Bar82].

The next subject that will be addressed in this paper is an introduction to Prolog, which is the programming language used to implement the prototype of the module development system. The reason for this inclusion is that I feel the programming language made a significant difference in the ease and clarity of being able to implement a system of this type. Prolog is a 'logic programming language', and so it has some unique features to easily handle the types of data structures that must be supported when dealing with partially developed programs [War79].

The final chapter is an in depth description of the module development system. It is hoped that this system can be used as a simple, clear teaching tool for software engineering students that want to study the subject of program transformation systems. The target programming language at the present time is Modula-2. The system is geared to develop library modules consisting of an Implementation part, and a Definition part. The system hopefully could be configured to work on other languages, a possible problem is that some rules that make up the system store knowledge such as the legality of identifiers, using block structured bindings, and the relationship between the two parts of the library modules of Modula-2. It would not be too difficult to replace target languages, the numbers of rules of this type are few, and the syntax of the

target language is part of the input to the program.

The module development system is a target-specific system, algorithms are specified in parameterized form by input grammars that are augmented by semantic actions. These 'grammars' are representations of algorithms in a production rule format. The scope of the library modules that the system develops grows as new grammars are written. It is my opinion that these grammars are as easy to write as programs, but once they are written and debugged, many modules can be developed by them. The particular instantiations can differ in types, sizes, and in particular procedures that the user may wish to put into the module. The identifiers the user wishes to EXPORT is also chosen by users when they develop a particular instantiation of a module.

The size of module that can be developed is not large, however typical Modula-2 library modules fit into this range easily. The system was intended to be a classroom teaching tool, but a program of this type could be a very useful software engineering tool. It extends the concept of Modula-2 modules to the flexibility and functionality of an Ada generic package, by making all the attributes of a module parameterized.

The system itself develops programs by taking the grammars, and parsing them, rewriting the program by

choosing rules to find productions for non-terminals found in the grammar, under the guidance of a user. It builds a program tree, with the right-hand-sides of the productions making up a subtree under the non-terminal nodes. Functionality includes expending the tree, deleting subtrees, if the user decides he wants e change, edding more code, such as an edditional procedure, again at the user's discretion, and writing versions of the module to disk files.

The system was quickly prototyped in Prolog. It was short end modular. Rules were added easily, when new needs for semantic ections were identified. If e specification of the program template was wanted in the databese, the grammars were written and dehugged easily.

1.3 CONCLUSIONS

Researchers ettempting to develop general purpose automatic programming systems are probahly not going to come up with very useful systems for a long time. Common sense epproaches should look at how human programmers approach software design. How do humans store programming knowledge, and attack designing software systems? A study of this should help develop automated tools to assist this process. Humans don't look at input end output specificetions end start making up a

mathematical function that maps the inputs to the outputs. A typical programmer thinks in terms of inputs and output data patterns, and from those specifications formulates a useful data structure that can model the information that must be processed by the system. The programmer knows of abstract data structures used to handle data similar to that needed for this particular system. The programmer then looks at the requirements, and break the task down into the subtasks that constitute individual functionality such as input, processing, and output of the data. The programmer recalls other procedures similar in functionality, and abstract from that particular solution the common problem solving steps, which is then customized to the particular implementation. Usually the actual algorithm doesn't even have to be remembered, just that it had a particular behavior, so instead of remembering the algorithm steps, a copy is made of an old program, and it is modified. The approach has a basic problem solving basis. Sometimes a problem presents itself that is novel, and a programmer will need to synthesize an algorithm to accomplish the unique code. This skill is what the general program synthesis systems are attempting to automate. This is a skill that people are very good at, though, and attempting to get systems to do this part does not seem as critical as automating the more mundane task of keeping track of several code

components and making sure they interface in a correct manner. Much of programming is not writing novel, and unique algorithms to solve problems, but recombining old solutions in different ways. People get bored doing this sort of thing, and very bogged down in the petty little details. I see a lot more promise in systems such as the Programmer's Apprentice [Wat82], which assists the programmer in keeping track of code details, and leaves the novel and unique problem solving to the human involved. Target program development systems that store skeleton templates of the code fragments that the programmer wants to pull out and use over and over can directly aid this typical human program development process.

Future research in the development of these systems include finding ways of storing these program fragments, so they can be easily called by users, perhaps specified by natural language. How to specify these code fragments is also a good question. There are three basic levels of 'knowledge' about software systems. Different metaprogramming systems incorporate this knowledge to different degrees. Syntax-directed editors have knowledge about the legal syntax of a programming language. They do not allow the development of a syntactically incorrect program. The next level is the programming language semantics. The most difficult knowledge to codify is pragmatic or intuitive

knowledge of programming tasks, that expert human programmers have an abundance of. Thus domain of knowledge is the HOW of programming. How is this type of knowledge represented? Should the different levels of knowledge be represented all mixed together, or in separate rules? How should the knowledge be represented? What size of chunk is a 'natural' representation for an abstract algorithm? How should programming language syntax and semantic knowledge be made known to the system? How 'smart' should the system be in combining these code fragments, i.e. can we get the system to be able to reason about the code fragments that it manages?

The answer to the software crisis is to automate the program development process as much as possible. Intelligent program development environments that can assist programmers are needed now and parameterized libraries of modules managed by systems that know about these libraries are an immediately useful tool. The prototype system developed to illustrate these principles help programmers create instances of library modules from abstract templates. The resulting modules are correct and compilable. If programmers want to extend the data structures and algorithms the system knows about additional code fragments can be specified. Tools such as these that manipulate and assist in the creation of software will soon be

indispensible aids for all serious software developers.

CHAPTER TWO : AUTOMATIC PROGRAM GENERATION SYSTEMS

2.1 INTRODUCTION

This chapter is an overview of current trends in automated software development systems. These systems are a very active area of research [Fre85],[Yeh85],[Bar82]. I will first explain what these systems are, and explain their underlying similarities. I will then explain the role of logic and artificial intelligence techniques in designing these systems. Finally, I will outline the basic categories of systems, and give specific examples of systems that illustrate the major approaches.

2.2 AN OVERVIEW OF PROGRAM DEVELOPMENT SYSTEMS

Programming language are notations used to describe data structures and operations. Programming language semantics are described in various different ways [Hor82]. Semantics should be described formally so there is no ambiguity in the meaning of a programming language statement. One formal definition is the description of a virtual machine. Define the set of operations and data structures that make up the machine, and describe the sequence of operations that executes when a program statement executes. The semantics of the rules is defined by a set of rules which show how the programs will be translated onto the

virtual machine. Other formal methods for formalizing semantics include the axiomatic and the denotational semantics notations. These formalisms specify rules which describe stata changes independently from a machine, as funotions that map from state to state. Tha notations describe a language hy rules which show stata change before and after exaoution of a programming language faature. The notation is mathematical logio, and an interprater or infarence engine is naeded to make it executable.

Computer programs execute operations to manipulata data structuras. Soma computar programs manipulata other computer programs. These software systems are known as metaprogramming systems [Cam84]. In order to do this, these systems have to read in a taxtual representation of a program, stora the program in an intarnal data structure, and when it has finishad manipulating the 'program' which is data, it then must reconvert it to a textual representation, and output the program. There are many differant ways to represent programs. A metaprogramming system chooses one or more ways to represent the internal representation of a program. A text aditor is a metaprogramming system, which operates directly on the textual representation of the program, with no knowledge of the underlying syntactic structure, or semantic rules that correspond to the program. Another level of

metaprogramming systems parse the program, and store the program in a form that represents the syntactic structure. Systems of this kind include syntax-directed editors, parsers, pretty-printers, and cross-reference generators. Syntactic structure is well-understood and formalized in terms of production rules or grammars. Programs can be analyzed according to their syntactic structure, by looking at the legal combination of 'sentences' according to the target language grammar. Rules that are needed are the legality of a sequence of tokens, choosing one of an alternation rule, and recognizing if a sequence of statements fits a rule that calls for repetition. The next level of understanding of a program structure, is to have rules about tokens stored in the program structure. These rules would define relationships between different parts of the structure, capturing the context-sensitivity of the language. Compilers that enforce typing must use information about previously executed declarations that is stored in a symbol table to check the legality of a construct. This is an example of type recognition. Context operations include what is the parent of this token? what is the type of this token? what is the next token, or the previous token? Being able to access this information directly supports the ability to manipulate the data structure in a meaningful way. Editing of the program structure

is facilitated, only meaningful operations can be done to the data atructure. Editing commands include Replace, Delete and Insert. When part of a program structure is Deleted, all of the information relating to the tokens that were part of the structure must be removed, such as local tokens that were in the symbol table. When a program structure is replaced, the contextual information about that part of the structure must be replaced also. Another transformation operation is an Instantiation, create a particular instance of a token, by getting a constant of the target language and associating it with a particular token.

Additional levels get increasingly difficult to represent, and to formalize. The next level should bave knowledge about programs forms, giving the system a vocabulary that corresponds to data and control flow. The system has a rule that knows a Loop needs an initialization, a body, and a terminator. This level is very general indeed, and cannot tell nonsense from meaningful program fragments. The next level is a representation of stereotyped program 'achemes', 'plans', 'patterns', 'prototypes' or 'abstractions' {as they are referred to in different systems}. Examples of these rules would be the way to combine code to get stereotyped algorithms that most programmers know about, such as bow to interchange two variahles, how to insert a value into a list, how to insert a value into

a hash table. This is a very high-level. This level associates a programmer's intentions with the code features that are needed for the implemention.

At this point we have lost track of the function of this embedded knowledge, is it to reason about program structures that we input to the system, or is it to generate programs (analysis vs synthesis) ? The nice thing about these systems is that the very same knowledge that allows analysis of a program, can assist in the generation of programs, hy using the rules differently.

The stereo-typed rules, or program schemes, that describe how to implement a particular program fragment is not a piece of actual software, but the two are related by parameterization. All of the ohjects in the program can he hound to particular values, and this is called 'instantiation'. This is how actual software fragments can result from these schemes. Program transformation systems that aid in program generation concentrate on a specific set of rules that translate program schemes. The input is a very-high-level program soheme, and the system goes through a series of transformation rules, making the developing program data structure more and more specific, until the program is in a different form, usually a more efficient target language. The transformation rules

preserve the semantics of the program description, and show valid replacements for pieces of programs schemes.

A program transformation system is a system that supports the design of software automatically. These systems store programming knowledge, and rules of using this knowledge in the creation of a particular instance of a piece of software written in a particular target language [Par83].

2.2.1 FORMALIZATION

Traditionel 'hand-crafted' methods are falling short of dealing with the 'software-crisis', thus the active research into automating software engineering. Software engineering methodologies are badly in need of formalization to precisely represent and reason about the process of constructing software systems. Logic, and AI (artificial intelligence) are two possible tools to aid in this formalization [Bar82], [Sus85], [Dar84]. Applications using AI techniques have mushroomed in recent years with the popularization and availability of knowledge-based expert systems. Knowledge-based programs are used in such diverse application areas as assisting doctors in patient treatment, geologists drilling for oil, and engineers trouble-shooting computers. The ability to apply knowledge to problem solving underlies these systems. Automatic program generation is a potential candidate for a knowledge-

based expert. It is an area tbat requires knowledge about a problem, and requires reasoning using tbis knowledge. In order to construct systems to automatically reason about programs, we must come up witb metbods to represent tbe knowledge, and apply it in tbe construction of programs. A programmer's knowledge has many levels of detail. Large software projects can be extremely complex. How can tbis information be described in facts and rules tbat are precise enough to be used by a machine performing programming tasks? In addition, tbe buman users of the system must be able to describe the specifications of the problem in 'natural' terms [Bar79], [Bar82].

2.3 THE ROLE OF KNOWLEDGE ENGINEERING

Knowledge-based representation has been defined as "a combination of data structures (declarative) and interpretive procedures (procedural) that, if used in tbe right way in a program, will lead to 'knowledgeable' behavior" [Bar82]. A database contains facts. A knowledge base is a database augmented by rules for applying and combining tbe knowledge, and an inference mecbanism.

One of the requirements of a useable notation for programming knowledge is tbat tbe rules and facts be precise and detailed enough that a machine could use

it in performing programming tasks. All of the background context knowledge that humans take for granted must be incorporated in the facts and rules. Application of the rules muat then be used for constructing concrete implementations of abstract algorithms.

The basic methodology involved in knowledge engineering is to express knowledge about the system'a task in a machine-useable form. A decision that influences the flexibility and generality is the amount of information to store in individual facts. By separating the information into relatively small, identifiable chunks, the flexibility of the system increases. The small piecea can be combined in many different ways. An additional benefit is the system's increased ability to explain it's own actiona. The difficulty of retrieving, combining and inferring higher-level functionality increases, however. It is a trade-off between flexibility and power.

2.3.1 TYPES OF ENCODED KNOWLEDGE

The programming knowledge that needs to be stored in the knowledge base consists of facts, and rules for dealing with these facts. The facts contain information about the syntax of the languages involved, contextual relationships between different data objects, other semantic relationships, abstract data structures, and abstract algorithms [Sus85]. The rules must

essociete the sementic denotations with program syntactic ohjects. Some examples of knowledge that the systems must be ahle to utilize are; for example, is X a suhrange identifier? Whet is the base type of the erray? make a temporary variahle to store an intermediete result of an expression; whet is the non-locel binding of identifier Q? Return the value of the lowerbound of A.

The rules that store the specific rules for programming language syntax, or procedures make up the rules of the system. These rules must be dealt with by other higher-order, {or trensformation} meta-rules. The specificetions will be transformed via the meta-rules using the specific basic knowledge rules.

There are two general forms of meta-rules [Per83], en elgorithm, or procedural rule, or an ordered pair of program replecement schemes (pattern replecement rule). The basic rules ere termed 'FOLD/UNFOLD' rules, which repleces a nonterminal with it's left hand side, or vice-versa. There ere suh-rules, thet work on particular attributes of the partially developed program data structure, such as consistency checkers, end identifier binding rules, and representational implementation details.

2.3.2 METHODS OF INFERENCE

The problem of software design can be broken down into two basic phases, the mapping of a human conceptual model of a problem to solve to a formal specification, and then the mapping of a formal specification to an efficient imperative programming language. Some systems attempt to automate part of the first phase, others input a formal specification and transform it to a target programming language.

The first phase has the task of gathering information from the user and integrating it into a consistent model. The model is usually based on a knowledge representation technique, such as a semantic network or frame heirarchy. The second phase has the task of translating the specification into another form, while preserving the semantics. This phase is much easier to do correctly, there are no holes to fill in, and no inconsistencies to resolve. Once the translation rules are formulated, the job of the inference mechanism is to select the correct translation rules to map step-by-step from the specification to the target programming language formulation. The inference mechanism is therefore determined by the structure of the rules, and the relationships between different rules. The rules must chain together in some way, which the inference mechanism can use to make a correct selection. The inference mechanisms in these systems range from purely deductive reasoning, to systems that mix

inference with some heuristics, to systems that select rewrite rules under the control of the user.

2.4 CATEGORIZING PROGRAM DEVELOPMENT SYSTEMS

There is a large variety of software engineering tools being developed to help automate aspects of the development life cycle of software. These tools range widely in functionality, and knowledge. The tools under investigation in this paper are described as being automatic programming systems. Within this category falls a widely diverse set of systems. The span can include everything from structure editors, to general-purpose deductive program synthesis systems.

Automatic programming systems assist humans in some aspect of program development, and they can be classified according to four characteristics: a specification method, a target language, a target domain, and an approach or method of inference [Bar82]. All of the components influence the choice of the others; for example, a program generation system that takes examples of a program's input and output as a specification will need an inductive method of inference to figure out the mapping function from the inputs to the outputs.

2.4.1 SPECIFICATION

A variety of approaches have been used for a specification method. The basic categories are natural language, specification by example, or formal specification.

2.4.1.1 Natural Language Interface

Natural language systems are characterized by engaging users in a dialog which specifies the behavior of the desired system. From the dialog, an internal model of the specification is constructed. The user is asked to fill in any information that is needed, or resolve any inconsistencies detected.

An example of a system built that developed programs in a limited domain based on natural language specifications [Bie84] acquired a model for queueing problems and translated the model into GPSS simulation code. The model was stored in the form of a semantic network. The system knew what it needed for a complete GPSS simulation, and asked the user if it's knowledge was not complete. The system has the ability to translate the semantic network information back into English for user verification. Since the internal data structure was a semantic network the user didn't have to input the information in any special order. The system generates programs with a high level of performance. The application domain is very narrow, however, queuing algorithms in GPSS.

Tha SAFE (Specification Acquisition From Exparts) system [Bal85] takes as input an informal description of the prohlem, and its solution in a natural language, parenthasizad for ease of parsing. Tha parsing results in a series of event descriptors, which are then translatad into procedure definitions. SAFE attempts to fill in missing operands, and do other intalligent guessing, but does ask the user when it gets stumped. The SAFE system synthasizes a formal model intarnally, and outputs it in the form of a formal specification, which is passed to a program transformation system.

Natural language programming is a very flexihla and powerful communication medium. Machines, however, are a tool daveloped to do repetitive and exacting tasks, and communicating with them must be in unambiguous form, so there is no confusion about what they are supposed to do. Since software systems may be specified in a narrow domain, tha ambiguity of natural language is a solvahle problem, and can be solved if the system can detect inconsistancies [Bie84].

2.4.1.2 Input/Output Trace Specification

Providing examples of input/output behavior is a specification tachnique intermediate in difficulty between natural language specification, and formal specification. The user providas input and output

examples, sufficient for the system to deduce what the program is to do, and the system constructs it. The synthesis method that works with specifications of this type must detect a repetitive computation on the input/output traces using pattern-matching techniques. A function must be synthesized that extends to infinity the recursion properties of the given input/output values [Jou84], [Sha84]. Irregular programs with many cases or that have diverse functionality can't be handled by these systems. These systems need a dialogue with the user also to resolve inconsistencies and give further information. This form of specification is interesting because it is easier to synthesize programs from input/output traces than from natural language specifications, and it is easier for users to specify programs in this manner, than in a formal specification language.

2.4.1.3 Formal Methods

The third form of specification is formal. The earliest attempts at automating parts of the programming process involved the development of languages such as FORTRAN, ALGOL60, and COBOL. The first compilers were hailed as 'automatic programming'. The goal was to use "abstract" operators and control structures whose implementation required many machine instructions. A natural continuation of this trend is the development

of even higher level languages, attempting to make the specification languages more and more 'natural', and also executable [Hor85].

Very-High-Level-Languages, and formal specification languages which are more declarative than imperative are being developed. Prolog (described in chapter 3) has been critically analyzed as a tool for the formal specification of several systems, including an Ada compiler [Ganzinger85]. Since the specification language is executable many errors and inconsistencies are caught at design time. This rapid prototyping has many advantages, but the knowledge of the domain must still come from an expert human, no matter how declarative or high-level the language.

2.4.2 TARGET LANGUAGE

The goal of an automatic programming system is to generate software. The programming language of the resulting code is the target programming language. Many of the experimental automatic programming systems had a target language of LISP. Now these systems are attempting to adopt to 'real' applications languages such as Ada, or Modula-2. The target languages of the 4th generation languages are database query, COBOL or PL/I programs.

2.4.3 TARGET DOMAIN

Knowledge of programming is very diverse and highly dependent on the application area of the programs Some of the systems are theoretically general purpose, but in reality can only generate a restricted subset of programs. Others store in the knowledge base explicit procedural knowledge of particular target domains. For these systems, as the knowledge base grows, the applications that can be dealt with also grows.

2.4.4 INFERENCE

2.4.4.1 Theorem Prover

Formal verification describes methods of specifying programs in terms of input and output predicates. The methodology consists of defining the language symbols and operations in terms of formal logic input/output predicates. For straight line oode; each operator should be assigned a set of verification conditions that can be operated on by a theorem prover. If the theorem prover succeeds in proving the conditions, the program has been verified. This idea has been applied to program generation systems. The input is the input assertions, and the output assertions, and the generated proof is a program that maps from the input assertions to the output assertions.

Input and output assertions are specified in the

predicate calculus, or some other formal proof method. A theorem prover then is asked to prove that for all given inputs, there exists outputs that satisfy the output assertions. The proof yields a program as a side-effect [Bar82].

One aspect of automatic programming systems is the user-interface. The ahility to specify the program formally is more difficult than writing the program. This seems to be a dead-end approach, unless a specification method that interfaces to the formal specification is developed to allow a more natural specification technique.

2.4.4.2 Transformation Systems

A program transformation system converts a specification or description of a program into a form that is closer to the target language while it preserves the semantics. Compilers fit into this category, however 'transformational' systems are usually characterized as being 'intelligent' about choosing different possihle transformational steps. As these systems become more intelligent, they use deduction, or heuristics to decide what transformations to perform on the developing program. The specifications are generally in such a high-level, that they are more of a suggestion of what the user would like, than an imperative. The output is generally a conventional high-level language.

Program transformation systems therefore form another virtual layer above the compiler layer of a typical system. (hardware monitor - high-level-language translator - program transformation system - natural language interface) [Par83].

Target domain transformation systems traditionally use algorithmic knowledge in the same format as the more general problem solving classic artificial intelligence problem solvers. Many of the initial attempts at program generation used this approach.

Problem solvers accept specifications in terms of initial and final states (NOAH, HACKER). The earliest work in problem solving involved determining a single sequence of operations satifying the input/output relation [Nil75]. Classical artificial intelligence goal-directed search for solving a problem form the basis of the inference mechanism.

If the idea of the plan-formation problem solving techniques are combined with knowledge-bases, the systems become more realistic, although theoretically less flexible. The design questions involved in a transformation system then becomes how to represent program state information, how to represent initial preconditions, how to recognize the correct state to state translation rules, and how to show a state change.

Programming knowledge must be encoded as goals, or problem requirements, or plans, methods for implementing goals. The program generation problem becomes one of finding a series of plans that transform the initial state into the goal state; with the resulting proof yielding a program. Transformation rules can be expressed procedurally, or as a pair of related program schemes, carried out by pattern matching and instantiation.

2.4.4.4 Knowledge Based Systems

The knowledge-based approach relies on the knowledge residing on the knowledge base itself, which can be added to, deleted from, or replaced. The flexibility and utility of the system is dependent on the rules that make up the system.

One major limiting factor is the amount of programming knowledge to which these systems have access -- it is much easier to understand a program if you know a lot about what it is supposed to do. Knowledge-based systems can perhaps answer this need. The central feature of these systems is that their performance is based , not on their application of a few general principles, but on their access to large amounts of task-specific knowledge.

2.4.4.4.1 Flexible Inference - Knowledge Base

The PSI system is a collection of knowledge-based experts that work together to synthesize LISP programs. The input to the coding expert is a formal specification language. The rules of the coding expert and the efficiency expert successively transform the abstract program description into an efficient implementation. The system is described in more detail in section 2.4.6. [Bar79]

2.4.4.4.2 Deductive Inference Knowledge-base

The DEDALUS system derives LISP programs automatically from input-output specifications in LISP-like mathematical-logical notation. [Par83] Program synthesis is achieved by viewing the specifications as a goal to be proved, and meaning-preserving transformations are applied deductively to transform the specification to equivalent LISP constructs. The transformations include knowledge about the programming language or programming techniques and rules expressing facts about subject domains. Adding new rules increases the subject domain of programs it can handle. [Par83]

2.4.4.4.3 User-Driven Knowledge-Base

An interesting system that uses the knowledge-based approach is the CIP (Computer-Aided, Intuition-guided Programming) project by Bauer and Samelson. The project specifies programs in an algorithmic

language called CIP-L, designed for transformational programming. It is a so-called Scheme Language, based on a treelike abstract data type, defined by algebraic semantics. The CIP transformational system is also specified in a heirarchical algebraic way. Any language can be handled by this system if it can be converted to a tree form, and back. The system handles program schemes, with context parameters. The transformation rules themselves consist of three schemes, so they are data to the system. The three schemes are the source template, the target template, and the enabling conditions. Rules can be applied, and also expanded, and composed together, thus basic rules can be combined into higher-level rules. The system does not automatically perform the transformation rules. Rules must be selected by the user.

2.4.4.4.4 Target-domain Knowledge-Base

The program development systems that contain knowledge about specialized target domains of classes of programs fit into this category. The knowledge generally comes in bigger 'chunks', which limits the flexibility but generally increases the power of the system to rapidly develop immediately useful systems. One such system that was described in [Par83] is called the TAMPR (Transformation-Assisted Multiple Program Realization) system. It was developed to help in the implementation

of numerical algorithms in various different machines, and software systems. The algorithms are stored as 'prototype programs'. Examples of transformations that the system can do include converting single precision to double precision, changing a two-dimensional array to a one-dimensional representation, and converting basic linear algebra subroutines to executable FORTRAN subroutines. The authors report that TAMPR has successfully implemented a number of widely used numerical subroutine packages.

2.5 ILLUSTRATIVE APPROACHES TO AUTOMATIC PROGRAMMING

2.5.1 PROGRAMMING ENVIRONMENTS - THE PECAN PROGRAM DEVELOPMENT SYSTEM

One step in the direction of automating the programming task is the use of programming environments that are knowledgeable about particular programming languages, and help the programmer debug and write programs. An example of a system that was developed to support an environment of this type ie the PECAN system developed at Brown University. The PECAN PDS supports multiple views of a user's program [Rei85]. The views can be representations of the program or of the corresponding semantics. The primary program view is a syntax-directed editor. Other semantic views include expression trees, data type diagrams, flow

graphs, and the symbol table. An important feature of PECAN is language independence. A PECAN program development system is generated for a particular language from descriptions of the syntax and semantics of the language. These descriptions are used to produce tables and code to direct the language-dependent modules of the system. Programs are stored as abstract syntax tree forests. Abstract syntax trees are used in many program development systems, since they are a convenient halfway point between syntax and semantics. The tree-handling module will answer queries, such as type of node, or arity, and will edit the tree, deleting, copying and expanding subtrees.

2.5.2 INTELLIGENT PROGRAMMING ENVIRONMENT APPROACH - THE PROGRAMMERS APPRENTICE

The Programmer's Apprentice is described by it's creators as being a Knowledge-Based Program Editing tool [Wat82]. This would correspond to having an electronic assistant or secretary, that could check for errors in your programs. In order to do this, the system needs some knowledge, which consists of a knowledge-base of program plans. The plans describe standard algorithms and data structures. The knowledge-base also contains basic programming knowledge, such as the syntax of the target language.

Input to the system is a textual rapresentation of a program. Tha analyzer attempts to match the program to one of it's stored 'plans'. If the system can't find a plan to match, it creates one to correspond to the program. In this way the plan library can be axtended. Othar components of the system are the coder, which converts the plan reprasentation of tha program back to target progrem text form, tha plan aditor, used to modify the plan reprasentation of the program, and the drawer which draws a graphical representation of tha plan raprasentation. The similarity of this system, and a transformational progrem development system is that a library of standardized program algorithms is usad to assist in assisting in the programming procass. Tha diffarance is in the way the knowledge is applied. Input is in the target programming language, not an abstract specification, but the P.A. gives all the assistance of a syntax-directed editor. Transformations ara under the control of the programmer, not the development system. That is why it's called an Assistant, and not an Expart. This is an extremely powerful and productiva system, it is mora flexihle than the code generators, because it is not limited to a restricted domain of programs. A programmer can rapidly build a program by referring to fragments in the library, and customiza it hy using the plan editor.

2.5.3 4TH GENERATION TARGET APPLICATION APPROACH - MARK-II PROGRAM GENERATOR

There are a number of fourth generation languages that have been designed to work with database applications that are fairly routine. One such language generates COBOL and PL/I programs. The authors term it a 'nonprocedural specification language', even though it is processed by a computer. The reason they didn't want to call it a programming language is that it describes data, and data relationships, but no procedural information. The processor for this language is called the Model II, which is a program generator. The authors feel that this is a significant system, because the elimination of procedural aspects of programming reduces the load on the user, and the ability to handle problems with complex input/output, database and reporting requirements is extensive.

The design is based on data descriptions, unordered presentations, implicit iterations, and system-user interactions. Unordered presentation is possible, because the Model II establishes the precedence relations between statements, and iterations are generated from other aspects of the specification such as specifications over repeated data, and the use of subscripted variables in specifications. Any

discrepancies, redundancies or errors are corrected by asking theuser if the system's assumptions are correct. The authors contend that the PL/I and COBOL programs produced are efficient and of comparable quality of human-coded programs. [Pry84]

2.5.4 SPECIFICATION BY EXAMPLE APPROACH - THINKPAD

ThinkPad is a program development system that uses the specification technique of programming hy example. Examples of program behavir are specified hy demonstrating operations on sample data. The system generalizes the demonstrated behavior for application with other data.

The system uses graphical editing as the equivalent of programming. A user demonstrates an example of what the program is to do, such as drawing a binary tree and demonstrating how insert and delete funotions are to perform. The operations can only be legal ones, and the visual transformation is semantically unamhiguous. [Rub85].

2.5.5 THE REUSEABLE MODULE APPROACH - PARAMETERIZED PROGRAMMING

The basic idea of parameterized programming is to maximize program reuse hy storing programs in as general a form as possihle. One can then construct a new program module from an old one by instantiating

parameters.

Tha system uses various functions to accomplish tha instantiation. It's basic building blocks ara parameterized modules. 'Theorias' ara used to defina properties an actual parametar must have for it to be substituted for a particular formal parameter. Module exprassions ara usad to modify modulas, by adding, deleting, or renaming functionality. The final function is the instantiation of a parameterized module to an actual modula instance [Gog84].

2.5.6 THE KNOWLEDGE-BASE APPROACH - PECOS

Program knowladge storad by the PECOS system is a codification of programming knowledge for many aspects of symbolic programming. The primary abstract concepts involved ara collections and mappings, along with operations and control structures. The principle representation techniques for sets are linked lists, or arrays, and the reprasentations of mappings are tablas, sets of pairs, and property lists. In addition to general symbolic knowledge rules, thera are about 100 LISP implementation specifio rules.

Tha easiest way to understand the system is to look at examples of rules storad by PECOS. They are actually storad in a formal specification language, hut Barstow expressed them in English to anhance under-

standing. The following are a few examples. [Bar79].

rule : a membership test on a stored collection may be refined into a test of whether any item in the collection is equal to the item being tested.

rule : a stored mapping with typical domain element X and typical range element Y may be represented with an association table whose key is X and whose value is Y.

rule : one technique for remembering the result of a computation is to save it as the value of a variable.

The above rules work for any language. In order to produce code in a particular language however, there must be rules dealing with that language. Knowledge about LISP is associated with the uses to which the LISP constructs can be put. Rather than describing the function CAR in terms of axioms or pre- and post- conditions, as is done in most automatic programming systems, the rules deal with specific uses of CAR, such as returning the item stored in a cell of a lisp list , thus there is never a need to search the knowledge base of facts about Lisp in order to see whether some function achieves a desired result. That information is stored with a description of the result. Searching is reduced, but so is the flexibility of 'inventing' new uses for the particular LISP function.

Lisp Rule: in LISP a program with name P, argument list A, and body B is written as a funotion definition with name P argument list A, and definition body B.

Lisp Rule: In LISP an assignment of a value V to a variable X may be implemented with a call to the function SETQ with X and V as arguments.

Lisp Rule: a test of whether an item is stored in some cell of a LISP list may be implemented as a call to the function MEMBER with the item and list as its arguments.

Refinement steps consist of adding properties to nodes, or replacing one node with another more detailed node. Query rules are used to get responses from the user to aid in program refinement steps.

2.5.7 THE TARGET-DOMAIN APPROACH - TRUE

TRUE {TRon User Environment} is a programming tool that generates real-time tasks from descriptions entered by a programmer. The authors [Shimizu & Sakamura] designed a real-time operating system called I-TRON for Industrial-TRON, TRON being an acronym for The Real-time Operating system Nucleus. The resulting programs are object code for I-TRON. TRUE creates a start-up task for system initialization, loads and starts execution of the generated group of tasks. TRUE 'tunes' or optimizes software for this system. TRUE uses a state-transition diagram to represent the structure of programs. This model represents what type of

events the system accepts, what action is executed in response to the event, and states the system would be in after completion of an action. The user of TRUE writes a program description based on this model. TRUE analyzes this specification program, and transforms it to optimize the code. After 'tuning' the program, TRUE compiles the program and generates tasks for the operating system. The authors are attempting to automate the process that human programmers execute when they look at a program, the analysis and decision of whether or not it satisfies requirements, including such things as meeting tight time and memory constraints. The description language that is the input specification for these programs allows users to specify actions, and to declare global variables, channels, and subsystems. Global variables are shared among subsystems, but TRUE utilizes semaphores to insure mutual exclusion. Channels are facilities for input/output, or synchronization between subsystems. An output statement is used to send a signal to a channel. Users also specify response time requirements, and TRUE evaluates the response time by simulation. Parameters of simulation are execution time of instructions, and system calls and frequency of event occurrences.

TRUE performs 'tuning' of programs by analyzing the program in the state-transition graph form, and

acquiring knowledge such as 'global variable X is read by task T'. 'local variable Y is written by action A'. State transition information can be changed into a list of facts that consist of state change information, assignment, input and output. The order of actions can be rearranged. The longest execution time is evaluated to get answers to response time. Strategies used in transforming programs consist of changing task priorities, decomposing tasks, and creating new channels, if it sees the need in increasing execution speed, it rearranges positions of actions, and attempts to shorten mutually excluded regions. TRUE was implemented in C-Prolog. The transformation rules are in the form of IF application conditions THEN transformation action. Some (simplified) example rules include:

```
IF (code pattern will match) "STATE S: event E -->
                action A; action B; Next S;"
                and action A is a target action,
                and action B is not a target action;
THEN decompose the code into two tasks.

IF "action A; action B;"
                and B must terminate before some
                task T can execute, and the user
                agrees that the order of some READ
                or WRITE operations can be changed
                for any global variable used by both
                A and B;
THEN replace the code by "action B; action A"
```

There are reportedly 10 to 30 rules for each class of transformations. Programs the authors have developed using this system the authors describe are a

communication program, {shown in article}, an alarm clock controller, a robot controller, and real-time games.

2.5.8 THE KITCHEN SINK APPROACH - ICON TEMPLATES

The Carleton Embedded System Design Environment (CAEDE) is a system developed to do the following: assist in prototyping embedded syetem software; assist in teaching students about designing such systems and incorporating design expertise in a programming environment, especially targeted at implementing communications protocols. CAEDE uses a graphical icon design interface to assist in the visualization of patterns and relationships while creating multitasking designs [Buh85]. The design is specified by selecting from menus of icons, either primitives or higher level icons. The output of CAEDE is a Prolog database of facts and rules which may be manipulated by Prolog software engineering tools that participate in the software development environment.

One of the tools that works on the Prolog database of design facts is an Ada code generator. The advantage of using Prolog for meta-programming liee in its ability to perform translations given only translation rules [Warren]. The code generator is a form of compiler. The rules are a fairly direct representation

in Prolog of Ada BNF syntax. The most interesting component of the code generator is the Rule-Based Structure Synthesizer. It contains the language syntax rules, end while attempting to go through them, must reference the design databese.

The euthors describe the system as a framework for relatively complete production of code from iconicelly entered design structures, resulting in an Expert's Assistent for rapid design prototyping of embedded systems.

The software knowledge in this system is based on e small number of concepts: etoms, ettributes, relations, constraints and actions.

Atoms: objects in the knowledge base associated with physicel objects of the software project.

Attributes: user-defined information associated with atoms.

Reletions: the heert of the SKB, a series of fects ebout the software project that is expressed es links between etoms.

Constraints: conditions on the relations and ettributes.

Actions: steps to be taken when e constraint is violated.

Software requirements are a very difficult subject to 'formalize', or make unambiguous. The euthors state that software requirements should involve e model of the real-world knowledge involved, in addition to

functional specifications.

2.6 CONCLUSIONS

The search to automate the programming task has been ongoing for twenty-five years. A suggested list for rating automatic programming systems [Bal73] includes the following:

- time and effort required (informality) for user, or how ambiguous and incomplete can the specifications be the system can handle?
- efficiency of design decisions, is any backtracking necessary?
- efficiency of the program produced
- reliability of the generation system
- computer resources (time, memory) required by system to produce a program
- the range and complexity of the tasks that can be handled by the system.

It is inappropriate to compare the systems based on these criteria without a 'test-drive', using representative benchmarks of some kind. It is also inappropriate to compare two systems that are not targeted to the same thing. For instance two systems that are targeted to produce embedded communications software could be compared critically. One aspect of these systems that can be compared is the target intention itself, and whether the system lives up to the

target intention.

The automated tools have increased in power and utility partly because of an increase in hardware capabilities, also because researchers are learning how to represent programming knowledge, and how to structure it in meaningful ways.

CHAPTER 3 LOGIC, PROGRAMMING, AND PROLOG

3.1 INTRODUCTION

PROLOG is a programming language, as well as a tool for application of automatic programming. The language was defined in 1971 at Marseille by Alan Colmerauer and Phil Roussel for applications in the natural language area. Prolog is more than a programming language, it is the first implementation of a language embedded in logic. [Cua85] A lot of interest has been generated in this language in recent years. This language is uniquely suited for program manipulation systems, but to understand why, a quick background of logic, logic programming, and the runtime environment of Prolog is necessary.

3.2 PREDICATE LOGIC

Logic was first devised in ancient Greece as a way to formally represent arguments. Logic is used to express propositions, or statements about the world. It also expresses relationships between propositions and how new propositions are formally and validly inferred from others. Terms are constant symbols, which name one object, or variable symbols which can be bound to different objects at different times, or compound terms. Facts are expressed as compound terms.

A compound term is a function symbol associated with an ordered set of arguments. Compound terms express relationships between possible arguments [Copi54].

Propositions about objects are the basic reasoning components of logic. Propositions are functions that map terms and term expressions to values of true or false. They are either compound terms, or they are constructed by combining compound terms with logical connectives.

The logical connectives are:

negation: ~A, if A is true, then not A (~A) is false, if A is false, ~A is true;

conjunction: A & B, (A and B) if both A and B are true, then (A & B) is true, else false;

disjunction: A v B, if either A or B is true, then (A v B) is true;

implication: A -> B, (A implies B) if A is true , then True if B is true. If A is not true, False;

equivalence: A = B, True, if A and B have the same value.

In order to talk about propositions, sets of individuals, and what is true or false about them must be expressible. Variables that may be arguments to compound terms serve this role, but in order to be meaningful, they must be quantified. There are two modes of quantification, the universal quantifier indicates that for all members of the entire set some predicate

is true; the existential quantifier states that at least one member of the set has the property.

example: Ax:man(x) --> mortal(x)

For all x, if x is a man, then x is mortal.
In simple English, all men are mortal.

Ex:man(x) & likes(x,quiche).
There exists some x, such that
x is a man and x likes quiche.

3.3 LOGICAL INFERENCE OF CONCLUSIONS

Logical inference is the procedure that is used to combine propositions in such a way, as to be able to come up with new propositions. In this way we can increase our knowledge. The earliest form of logical inference is called the syllogism. Aristotle described this form of reasoning. His most famous example is the mortality of man. All men know that they are going to die. How do they know this?? A person is not born knowing he is going to die. First you must have facts. One of them is that you are human. The second fact is the observation that all humans to date have died, so a fact of observation is that if you are human, that implies that you are mortal. From these two facts, that you are human, and that humans are mortal, you can come up with the knowledge that You are Mortal.

Logical Inference to prove the fact that someday we

will die:

```
Human(x).
Human(x) --> Mortal(x).
therefore, Mortal(x).
```

Classical logical syllogism, in a general form states:

```
A:X (p(X) -> r(X))    ; everything with property p
                        has property r.
A:X (r(X) -> q(X))    ; everything with property r
                        has property q.
Therefore:            ; therefore,
A:X (p(X) -> q(X))    ; everything with property p
                        has property q.
```

```
syllogism - classical forward-chaining inference
                    given:
                      A
                      A --> B
                      B --> C,Q
                      C --> D
                      Q --> R
                    result:
                      B,C,Q,D,R
```

3.4 CLAUSAL FORM OF FIRST ORDER LOGIC

A different notation for expressing first order logic relations of predicates is called the clausal form of logic. This form is not as 'natural' as the predicate calculus, hut all statements of first order logic can be expressed in clausal form. There are precise rules that relate the clausal notation to the predicate calculus notation. The following rules informally explain them.

Any existentially quantified variable is given a name, and treated as a constant, therefore, any variable that appears in clausal form is universally quantified.

Conjunctions are listed sequentially, and separated by commas. Inferences are reversed. The following examples illustrate these conventions.

CLAUSAL FORM	PREDICATE CALCULUS FORM
female(cory),pres(cory).	E:X(female(X) & pres(X))
alive(X) IF has-pulse(X).	A:X(has-pulse(X) -> alive(X))

The following is a rule for proving Y is an ancestor of X. Either Y is a parent of X, or there exists an X who has an ancestor Z, and Y is an ancestor of Z. If this is true, it implies that Y is an ancestor X. The predicate logic formula follows:

A:X,Y,Z (Par(X,Y) OR (Anc(X,Z) & Anc(Z,Y)) -> Anc(X,Y))

This is the clausal logic representation, the two sides of the disjunction appear on separate lines:

```
Anc(X,Y) IF Par(X,Y).
Anc(X,Y) IF Anc(X,Z),Anc(Z,Y).
```

3.5 LOGIC PROGRAMMING

The the inference mechanism of backward reasoning, which states "Conclusions If Conditions" can translate the logical inference steps into procedures in the following manner. Consider the following sentences the database of known facts.

All humans are mortal.
Socrates is human.

These facts expressed in clausal form would be:
mortal(X) if human(X).
human(socrates).

The goal is to prove that Socrates is mortal. Using the backward chaining form of inference, break the goal, or the fact you wish to infer down to sub-problems to solve. This is the procedural interpretation of the above logical statements:

1) To find X which is mortal, find X which is human.
2) To show Socrates is human, do nothing. (this was expressed as a fact).
3) To find X which is human, let X = Socrates.

Once we substitute Socrates for the variable X, using procedure 3, we have an X which is human, therefore X is also mortal due to procedure 1. [Kow85]

Alternative forms of logical inference:

syllogism - classical inference	resolution-
forward-chaining	backward-chaining
given:	given:
A	B.
A --> B	D <-- A,B,C.
B --> C,Q	A <-- F,G.
C --> D	F.
Q --> R	G.
result:	C.
B,C,Q,D,R	result: F,G,A,B,C,D

Consider one more example, the clausal form from above defining the ancestor relationship. Assume there are additional clauses in the database:

```
Anc(X,Y) IF Par(X,Y).
Anc(X,Y) IF Anc(X,Z),Ano(Z,Y).

{ these clauses are facts }
Par(sally,jim).
Par(jim,bart).
```

Is bart an ancestor of sally?? The first alternative that could be used to prove this statement is: Is bart a parent of sally? There are no facts or rules that can be used to prove this is true, so try the second alternative. Sally has provable ancestor, her parent, jim. Jim has a provable ancestor, his parent Bart. By the second clause, the clause can be proven by substituting {sally=X, bart=Y, and jim=Z}. The proof steps are as follows (see fig. 1): To prove

Anc(sally,bart), first see if Par(sally,bart) is true. The search for this fact fails, but there are two ways to prove this ancestor relationship, so next try the alternative rule, Anc(X,Y) IF Anc(X,Z) , Anc(Z,Y). Now the proof is 'to prove Anc(sally,bart), first prove Anc(sally,Z), and Anc(Z,bart)'. We now have two subproofs. The first is to find an ancestor of sally. This is stated as follows: To find Z which is an ancestor of sally, find Z which is a parent of sally. This is rule one, Anc(X,Y) IF Par(X,Y). In the list of clauses we have a fact, that the parent of sally is jim, Par(sally,jim). Therefore, to find a Z which is a parent of sally, let Z=jim. This substitution is made possible because of the fact, Par(sally,jim). Therefore, we have proved a new fact, Anc(sally,jim), which concludes the first subproof. Now, what remains in the overall proof, is to prove the second subproof, which is now Anc(jim,bart). The values of Z must be the same in both subproofs, for a simultaneous proof. Again in this proof, there are two ways to prove an ancestor relationship. Again, choose the simple rule, first, to prove Anc(jim,bart), prove Par(jim,hart). This is a trivial proof, since Par(jim,hart) is listed as a fact, therefore no action must be taken, it is given.

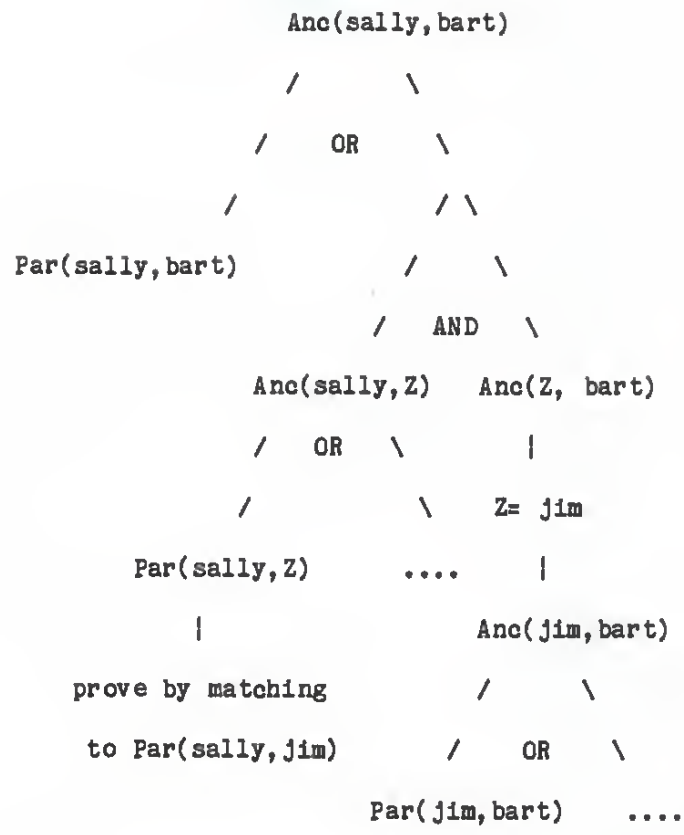

Figure 3.0: to prove the root is true, one branch of the OR branches must be true, all branches of the AND nodes must be true.

Proof: Par(sally,jim).

Anc(sally,jim) IF Par(sally,jim).

Therefore: Anc(sally,jim).

Par(jim,bart).

Anc(jim,bart) IF Par(jim,bart).

Therefore: Anc(jim,bart).

Anc(sally,bart)

IF Anc(sally,jim) AND Anc(jim,bart).

Therefore: Anc(sally,bart).

We have proved that hart is the ancestor of sally, using backward chaining.

Procedures are specified by logical clauses of the form: B if A1, A2, ... An. The As are the joint conditions of the clause, B is the conclusion. The procedures can be read in English in the following way: If there are values to match all variables in A1 ... An simultaneously, then B.

There is a rule of inference used to prove theorems from a set of logical statements in clausal form, called the resolution principle. This is the formalism of backward chaining inference. It states that if there is a substitution possible from the given clauses, for a given goal, then the proof procedure will find it.

3.5.1 TERMINOLOGY

A term in logic programming is a symbolic representation of a term in first order logic, it is a constant, a variable or a compound term. A compound term is a functor and a sequence of ordered arguments. A substitution is a set of pairs, (X -> t), where X is a variable and t is a term. In the above examples, where we let X equal Socrates, or let Z=jim, a substi-

tution was performed. For a given substitution theta, and term S, the result of replacing each occurrence of a variable Xi by ti , Stheta, is called an instantiation of S. Theta = {X1 -> t1, X2 -> t2, ... Xn -> tn}. Theta is a called a unifier for two terms S1, and S2 [Sha84].

3.5.2 COMPUTATION

Rules are expressed as goals, which are solved by first solving the subgoals. A representation of the rule is Goali = Subgoal1, Subgoal2, ... Subgoaln. If there is a clause and it is 'unifiable' {it can legally match} with theta, and C has a series of subgoals, {C = Sg1, Sg2, ... Sgm}, then the new goal that must be solved is Goali+1 = {Sg1, Sg2, ... Sgm, Subgoal1, Subgoal2, ... Subgoaln}, which was derived from Goali, and C with substitution theta. To 'prove' goal Goali with logic program P, you must have a set of triples: <Goali, Ci, thetai>. Goali+1 is derived from Goali and Ci with substitution thetai. As long as goals are replaced by subgoals, the system must continue to search. Eventually the goals must either match with a fact (has no subgoals), or it can find no clauses with which to match, thus the proof (or computation) fails. Logic program clauses have three possible forms:

1) A <- B1, ... , Bn	Procedural clauses, or rules. A clause with a head (goal) and a body (subgoals).
2) A <-	assertion {fact} A clause with a head, and no body, therefore if this head matches a subgoal, it proves the subgoal, a fact is true.
3) <- B1, ... , Bn	refutation, {question} A list of subgoals that will be taken as questions that need to be proven sequentially.

It must be shown that there exists values for all variables appearing in B such that B1 through Bn are all simultaneously satisfied (true). The proof procedure is a refutation procedure called resolution which tries to derive the empty clause from these clauses. If a refutation (question) is not consistent with a given set of clauses, the resolution proof procedure will find it.

Given a resolvent (question) when attempting to resolve it with another clause one of the clauses from a set must be chose. Given a choice of literal to be resolved upon there may be several A to unify with the chosen literal. Given these two choices, at each step in the proof, the refutation proceeds until it fails or succeeds. If it fails, another choice must be tried. Therefore the proof procedure assumes exhaustive combinatorial search of all the possible choices. According to logic it doesn't matter which clause is chosen [Kow72].

3.6 PROLOG (PROGRAMMING IN LOGIC)

In Prolog, programs are made up by a list of clauses. Clauses have two forms, they can be facts or rules. A fact is made up of a compound term, a funotor and it's arguments. A rule is made up of a single compound term as the head of the list, an IF operator, and a body. The body is made up of a sequence of compound terms. All clauses are terminated by a period. Uppercase letters denote variables, lowercase letters denote atoms. Variables can be unified to terms of arbitrary complexity. In this way data structures can be represented. For example to specify a sorted binary tree in Prolog, specify the relationships between the tree, by defining the structure in terms of relationships.

```
/* finds duplicate, or inserts the node */
handle-tree(Node, tree(_,Node,_)).

/* traverse tree */
handle-tree(Node, tree(Left,Node1, _))  IF
          less(Node,Node1), handle-tree(Node,Left).

handle-tree(Node, tree(_,Node1, Right))  IF
          greater(node,Node1), handle-tree(Node,Right).
```

Prolog also provides a list data type. Square brackets denote grouping atoms into lists, with the bar character '|' denoting ththe separation of the first atom of the list from the rest of the list.

```
List Notation:   empty list  []
                 [Head|Rest]

unifying [A,B|C]   with [1,2,3,4]
results in A=1, B=2, C=[3,4]

unifying [X|Y] with [q]
results in X=q, Y=[]
```

Here is an example of a Prolog program that has three arguments. All three arguments are lists, the third argument is the list that is the concatenation of the first two lists.

```
concatenate([], X, X).
concatenate([X|L], Y, [X|Z]) IF  concatenate(L, Y, Z).
```

Prolog programs can be read in two ways. A logical interpretation, and a procedural interpretation is possible. Interpret the concatenate procedure using a logical interpretation in the following manner. The first clause { concatenate([],X,X) } reads: the concatenation of the empty list to any X yields X. The second clause { concatenate([X|L],Y,[X|Z]) IF concatenate(L,Y,Z). } reads: the list beginning with any first member X, and remainder of list L, concatenated with list Y, yields a list that begins with X, and has a remainder Z IF list L concatenated with Y yields list Z.

Because this definition is not a function, but a description of relationships between terms, the pro-

cedure can be thought of in two ways. It can be thought of as a procedure that takes two lists and produces a third list that is the first two concatenated, or it could be thought of as a procedure to tell whether or not two lists are the result of a concatenation. In other words, depending on whether one tries to match the patterns in the heads of the clauses with atoms, to test for a match, or with variables, to generate possible answers, these clauses can use their arguments as either input, or output parameters. This ability is called invertability.

The logical or declarative definition of concatenate describes a static relationship between terms, it does not procedurally describe how concatenate should be computed. A knowledge of the operational semantics of the Prolog interpreter is necessary to read and understand the procedures with respect to how they are interpreted.

One goal of implementing a logic programming language was the perception that it was desireable to describe computation by declaring the specifications, and not showing the control of the implementation, let the underlying machine implement the specification language [Kow85], [Cua85], [Fer81], [McC84], [Clo84]. In practice however, in order to write efficient programs, the underlying operational semantics must be

kept in mind, otherwise the program will be inefficient, in the best case, and it will not work correctly in the worst case. This is true, because the subgoals that make up a goal must be executed in some kind of order, and the choice of which clause to unify a goal with must also be decided based on some kind of criteria. Since the implementation is being carried out on a sequential machine, these constraints must be enforced. Also, there must be the ability to have functionality in the Prolog system that is not defined for logic, such as input, output and arithmetic. To interpret the concatenate clauses procedurally, one must view the clauses as rewriting rules, with left-hand sides being rewritten in their right hand sides whenever the left hand side matches an occurrence of a query. View the clause head as the procedure heading, and the right hand side as the body of the procedure, that consists of a series of sequential procedure calls. (subgoals).

The interpreter builds an AND/OR tree, with OR nodes corresponding to different possible clause headings, solving the same problem in an alternate way; the AND nodes corresponded to the bodies of the procedures. The interpreter chooses one of the OR node paths, and sequentially executes each AND node, by finding a match of the subgoal to a clause head, and placing the subgoals of the body on top of the goal stack. If the

interpreter cannot match one of the goals, then a back-tracking algorithm goes back to the last choice point and explores that alternative.

The Prolog interpreter uses a depth-first search approach for the first solution, completely solving the first solution, completely solving the first branch of an AND node before going on the next branch, to find a second solution.

3.7 PROLOG - A RULE-BASED INFERENTIAL SYSTEM

3.7.1 A SIMPLE EXAMPLE

The following example is a dating-service rule-base system. It is a Prolog program, made up of facts and rules. To run the program, you give the program a goal to solve, such as: Who is a good match for John? Would Debra and Bob hit it off? The system tries to find answers using it's database of facts and rules. If you ask this program to generate all the matches it can it returns the following three pairs: Laura and John, Debra and Bob, Debra and Mike. There are no other suitable combinations.

```
/*     rules     */
/* find a couple who share a common interest  */
/* and do not have conflicting personalities. */

date_match(X,Y) :-     woman(X), man(Y),
                       shares_interests(X,Y),
                       not conflicts(X,Y).

shares_interests(X,Y) :-   enjoys(X,Activity),
                           enjoys(Y,Activity).

conflicts(X,Y)  :-     is_a(X,Type1), is_a(Y,Type2),
                       opposite(Type1,Type2).

/* order is important, so to show a relationship  */
/* that is unordered, both orders must be shown   */
opposite(X,Y)   :-     opp(X,Y).
opposite(X,Y)   :-     opp(Y,X).

/*     facts     */

opp(liberal, conservative).
opp(socialite, homebody).

man(mike).
man(bob).
man(john).

woman(laura).
woman(debra).

enjoys(laura,movies).
enjoys(laura,conversation).
enjoys(debra,sports).
enjoys(debra,movies).
enjoys(mike,sports).
enjoys(mike,conversation).
enjoys(bob,movies).
enjoys(bob,sports).
enjoys(john,conversation).

is_a(mike, conservative).
is_a(debra, socialite).
is_a(laura, homebody).
is_a(laura, liberal).
is_a(bob, socialite).
is_a(john,liberal).
```

To find a match using this rule-based system, you would present the query - "?-date_match(X,Y)." The

system would respond with :

X = laura
Y = john

Differently posed queries can give more specific answers, if you fill in constants instead of variables for the query arguments. For example:

"?-date_match(laura,mike)."

The systems response:

no.

3.7.2 EXAMPLE TWO: A COMPILER

On a more serious note, it is instructive to look at how rule-based inferential or 'expert system' technology has been used in systems that were targeted to solve real problems in software productivity in a production environment. Logical proofs, that are mechanized behave as problem solvers, finding a path from a goal condition to a resolution, by applying the correct rules in the correct order. A large number of problems can be expressed as path-finding problems, by trying to solve a goal that finds a path from an initial state, to a goal state.

The Intermetrics company was commissioned to design compilers to generate code for the space-shuttle on-board computers. The code quality had to be very good. They succeeded but the compiler was extremely

complicated and very difficult to maintain. The next time they contracted to design a compiler of similar quality they decided to use expert system technology. The benefits were the ability to describe the source language and the target language knowledge separately from the general knowledge of how to translate languages. The problem knowledge was partitioned much more cleanly thus the design was easier, as programmers came up with special cases that required clever instruction manipulation, they just added the rules to the rule base. The compiler was much easier to maintain and upgrade, enhancements could be added without having to delve into understanding the internal workings of the compiler. Their code generator works on an intermediate tree representation of the program. It uses pattern matching to replace subtrees with object code. It uses knowledge expressed as production rules to search for locally optimal code sequences. The rule's conditions are conditions that are tree templates that match, and actions that generate subtrees of lower level code. Alternatives are evaluated, adding to the intelligence of the system. [Fre85]

Transformation rules are used when particular enabling conditions are true.

3.7.3 EXAMPLE THREE: STATE TRANSITIONS

Prolog is uniquely suited to specify program transformation rules. Stanley Lee of GTE laboratories [Lee85] used a general state-transition framework for a rule-based software prototyping tool. The specified system's behavior is specified by pattern-oriented rules containing pre- and post-conditions for each transition. All of his specification models are written in Prolog and executable. The examples given were for a finite state machine, a database, and a Prolog interpreter. The general state-transition framework is defined as the set of system states, a set of inputs, a set of transition funotions (state X input) -> state; and an initial state, S0. A system is specified by specifying a set of transition functions, along with their pre-conditions, to enable the transition and post-conditions which hold following the transition. The definition is written as a Prolog relation: tran(Preconditions, TransitionName, Postconditions). To execute the system, the transitions must be fired, by the following rule:

```
fire(CurrentState, TransitionName, NextState)  IF
      tran(Pre, TransitionName, Post),
      satisfies(CurrentState, Pre),
      modify(CurrentState, Post, NextState).
```

The database example he specifies is modeled as a Prolog list of terms, [T1,T2,....,Tn]; the list represents the contents of the database, and the

members represent tuples. The transitions correspond to legal manipulations of the database. is_in(X), not_is_in(X) specify conditions of the database before a transition {legal transaction}, and also post-conditions, or results of tbe change the transaction causes. This system does tbree transactions, hires, transfers, and fires employees.

```
/*  HIRING TRANSACTION */
tran([not-is-in(works-for(Employee,AnyManager))],
      hire(Employee,Manager,Salary),
     [is-in(works-for(Employee,Manager)),
      is-in(earns(Employee,Salary))]).

/* TRANSFER TRANSACTION */
tran([is-in(works-for(Employee,OldManager))],
      transfer(Employee,OldManager,NewManager),
     [not-is-in(works-for(Employee,OldManager)),
      is-in(works-for(Employee,NewManager))]).

/* FIRE TRANSACTION */
tran([is-in(works-for(Employee,Manager))],
      dismiss(Employee),
     [not-is-in(works-for(Employee,Manager)),
      not-is-in(earns(Employee,Salary))]).

initial([works-for(alan,dave), earns(alan,20),
      works-for(mary,dave), earns(mary,25),
      works-for(bill,sara), earns(bill,30)]).

satisfies(State,[not-is-in(Term)|Terms])  IF
      not member(Term,State), satisfies(State,Terms).
satisfies(State,[is-in(Term)|Terms])  IF
      member(Term,State), satisfies(State,Terms).
satisfies(State,[]).

modify(State1, [is-in(Term)|Terms], State2)  IF
      modify([Term|State1],Terms,State2).
modify(State1, [not-is-in(Term)|Terms], State2)  IF
      delete(Term,State1,State),
modify([Term|State1],Terms,State2).
modify(State1,[],State1).
```

[example from [Lee], figure2, pg 212]

3.8 TEMPORAL LOGIC

By carrying the data structure through the rules explicitly as parameters, as the above example does, the system can become very large, copying the entire structure whenever a change of state is made. An alternative to this representation is to represent the state as a series of simple relations that reside in the database, and are referenced by the transformation rules. This introduces a problem, however, that is not addressable by a system defined to view the world as first order predicate logic does. In logic, if there is a fact or rule in the database, it is always true. The rules and/or facts that describe the state of the system, however must change when the system state changes. By using this method the system must be more powerful than first order predicate logic, it must have the ability to change the rules and facts of the system as the state changes during execution. Prolog has the ability to add new rules to the database, and delete old ones, thus it is more powerful than first order logic. The Prolog clause that adds a rule is assert(rule). The Prolog clause that removes a rule is retract(rule).

3.9 SUITABILITY FOR METAPROGRAMMING APPLICATIONS

It is no accident that many metaprogramming systems are implemented in Prolog. It has unique capabilities to specify transformation rules, being a

rule-based system [War80],[Gan85]. In writing a program transformation system, a program is input in one form, transformed internally, and output in another form. The rules for correct translation from one form to another must be formulated in the design of such systems. The structure of the source language, and higher-level combinations of the source language need to be formalized. For each form, a rule to translate it into another, more specific form must be formalized. These rules form the specification of the transformation system's function. Logic in general and logic programming aims at separating logic from control, it is tempting to take logic as a specification language. What makes it even better in the case of Prolog, is that the specification language is the implementation, since Prolog is executable.

Metaprogramming systems are dealing with complex data structures such as symbol tables, and program fragments in various stages of development. Rules that relate pieces of the programs to capture context sensitive information need to be stored. The ease of introducing and manipulating complex data structures, the unification process (matching actuals to formals on invoking a rule) that constitutes built-in pattern-matching permits complex assignment and selection of data structures resulting in simple, elegant programs. Prolog code is about one-tenth that of a comparable

program written in Pascal.

The very simple structure of Prolog makes it easy to generate standard or families of programs from a small database that describe the application dependent features.

For a particular instance of a program, these rules must be created that store this context-sensitive information. Thus the system must have rules that generate new rules for particular instances of a program. These rules that create new rules again step outside the realm of first order logic. Again Prolog has the functionality to create and dissect rules, thus it handles higher order logics. In chapter four, I will describe the Module Development System that I wrote, and show how I used this functionality to create program modules.

The construction of a system that operates on a data structure that is a program can be designed in the following way: at the top level, a specification is input, output is an efficient compilable source program. The interpreter system is a black box. To design the interpreter, first it must be decided how the input is related to the output. The structure of the specification is examined and rules for translating the semantics of the specification to a program are developed. These rules form the specification for the

interpreter's function. The next step is to implement the procedures to efficiently carry out the interpretation and rule-applications, in accordance with the specifications.

In a conventional imperative language this step is the most lahor intensive and error-prone step. Prolog this final stage is much easier, the specification rules ARE the implementation. They merely must he tested and debugged. Thus the procedures of the interpreter consist of clauses which are rules for describing a possihle translation of a particular construct of the source specification, and information from the user, to produce a particular instantiation of a target program.

The benefits of the close relationship between the implementation and specification include a more high-level, thus readahle implementation, a more easily proven correctness for the transformation rules. Also, additions and modifications to the specifications, such as adding new semantic actions are easily incorporated, as it consists of adding one more clause to a semantic action conversion rule.

A system such as this in another language would necessarily have to explicitly create and handle complex data structures, such as a symbol table, and the program development tree. All of the usual hazards of

constructing, handling end referencing these complex struotures would have to be explicitly coded.

In Prolog, the Prolog run-time system hendles deta construction and selection, end control operetion, actually much of the 'implementation' responsibility.

Compiled Prolog is compareble to compiled list or record processing languages [War80]. Space management can be a problem. The Prolog system automatically classifies variables into two types, 'locel' end 'glohal'. Local storage is recovered automatically when a procedure is done, and popped off the stack, unless the procedure is saved for possible backtracking. A second stack keeps track of hacktrecking choice points, and when they are exheusted, recovers all storage. Really large tasks with many suhgoels end levels of recursion cen exheust main memory quickly when each goal is matching with large deta structures. Therefore the system can only realistically handle small program modules in the development tree one et e time.

Prolog is e good medium for software implementation where the main priority is to implement a correot system quickly, or where the specifications ere changeahle or not entirely known. Better performance can be ohtained from lower-level langueges, hut Prolog implementations can serve as useful prototypes. With all of it's promise in the fields of software engineering

(prototype, specification, and program transformation language), and artificial intelligence (rule-hase, pattern matching), there is a tremendous research effort on finding better implementations and extensions of Prolog, to increase it's power, and it's implementation efficiency. [Carlson, Pittomvils et.al]

3.10 CONCLUSION

Logic programming languages have become noted in the areas of expert systems, datahase management systems, natural language processing, and software engineering. Since they are declarative in nature, they are a clear and concise notation for expressing hoth the specification and implementation of a program. The mathematical basis makes the languages easy mediums for manipulating specifications into implementations through transformation rules that contain and preserve the semantics.

The goal of a rule-base system with an inference mechanism such as Prolog for software engineera is to automate the software development cycle. A lot of the experimental synthesis approachs are theoretically interesting but show little promise for immediate solutions. Knowledge-hases that contain pertinent information, that is not fractured into the most elementary forma may not be as flexible hut show immediate applicability to aid in software development. With environ-

ments of this type, much of the design of software systems will still need a programmer's guidance to formulate the specifications, and guide the transformation process. Prolog is a powerful tool for implementing a programming environment where machine assisted transformations guide programmers, that are supported by intelligent programming knowledge-bases that understand the developing program more completely than typical environments of today.

CHAPTER 4 : A MODULE DEVELOPMENT SYSTEM

4.1 INTRODUCTION

In this chapter I will describe the module development system that I wrote to illustrate some of the principles of automatic program generation. The system was written in DEC-10 C-Prolog on a VAX-11/780 super-minicomputer at Kansas State University [Bow84]. The program is a target-domain system that generates library modules with a target language of Modula-2.

The module development system uses a translation grammar that incorporates semantic actions with the terminals and nonterminals of the productions. The system is target-domain oriented, having a set of module templates that are encoded in grammar form from which modules are developed. Fundamental code constructs for developing library modules for standard data types were identified and used as the basis for the grammar library. The system generates Modula-2 modules by applying the productions on the grammars, and executing the semantic actions under the direction of a user. The grammar is parameterized by the semantic actions, so the modules may be tailored to the individual needs of the user.

4.2 TARGET LANGUAGE: MODULA-2

One of the distinguishing characteristics of

Modula2 over it's predecessors is the module. A module designates a closed scope for identifiers that are declared inside the module. Any outside identifiers must be IMPORTED, and any inside identifiers used outside must be EXPORTED. This closed scope allows the formation of abstract data types, by allowing a module to be developed that describes the structure and operations on the type. What would be even more flexible is if the abstract types were themselves fully parameterized, so that they could be tailored to any specific needs [Wir79].

This is exactly what the Module Development System is designed to do. The target language domain is the production of Modula-2 library modules to implement abstract data types. The programming knowledge lives in a set of grammars that exist in separate input files. They are read into the system on demand. The grammars encode the syntax of Modula-2, as well as context-sensitive semantic and procedural information of the system.

4.3 PROGRAMMING KNOWLEDGE

The knowledge we need to represent is the syntax, semantics, and pragmatics of the programs that we wish to manipulate as the central data structure of the system [Bar85].

Programming environments encoda tbis knowledge to different degrees, divisibla into tbree distinguishing levels. Syntax-directed aditors hava knowledge of tbe syntax of programming languages. Semantics can be incorporated by making cbecks for context-sensitive attributes, such as no two identifiers declared in one block can be tha same. Most Difficult is pragmatic or intuitive knowledge of tbe programming task tbat buman programmers bave suob an abundance of. When you ask a buman programmer "bow did you solve that problem" or "bow did you write that procedure", be responds with tbe use of pragmatic knowledge. The reprasentation of this type of knowledge is vary difficult indead. Researcbers bave respondad with many answers. The traditional answer is to catalogue the different procedures in a library of subroutines tbat can be customized by band when writing the new program. Painstaking rules tbat specify tha knowledge of correct program transformations ara another method. Tbis mathod attempts to duplicate the buman programmers reasoning functionality. This method subsequently is muob more complex, limited and fraught with errors. A third method that lies somewhere inbetween is a transformational grammar. Tha grammar is more flexible than a library of alraady coded subroutines, becuase it is not a subroutine but a template for many subroutines that follow a specific pattern. Information about specific

algorithms is stored in a file of grammar productions, and semantic actions fully parameterize the grammar.

4.4 SYSTEM OVERVIEW

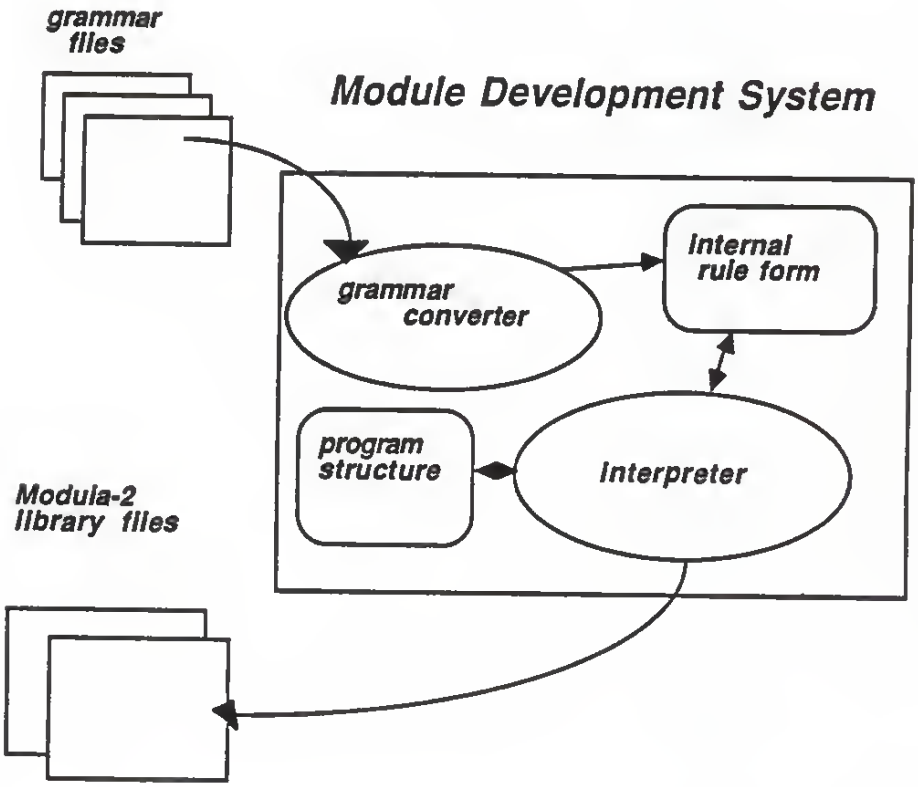

Figure 4.0: Overview of the Module Development System. Input is in the form of human readable grammars that must be converted into Prolog fact and rule form. Once in the form, the interpreter uses the Prolog form of the grammar, and builts the program tree. Once the tree is complete the system can write the completed Modula-2 module file into an external file.

The system has two main components, the grammar converter, and the interpreter. Grammar files are loaded by the system after being selected by the user. Different library modules are stored in different grammar files. The user must wait while the grammar is converted, then the interpretation phase begins. At this phase, the user is queried to enter information, such as identifier names, and operations, or data types desired. User's then can view the program so far. They can add additional operations, or they can undo some previously done action. At any inspection time, they can write the program out to a disk file.

The interpretation of the grammar is the heart and soul of the system. The interpreter reads through the grammar processing each symbol. Terminals are not processed, they are passed by untouched. Nonterminals become the root of a tree, the branches become the right-hand-side of the production rule that the nonterminal heads. If there is a choice of which production to choose, the user is presented with a menu from which to make a decision. When semantic actions are encountered in the grammar structure, they are executed. They result in an action such as getting an identifier from a user, retrieving from the database an already recorded identifier, or choosing a produotion.

4.5 GRAMMAR CONVERSION

The inputs to the system are in the form of a special form of grammar that is augmented with semantic actions. The grammar is read in by the grammar conversion portion of the system and translated into internal Prolog form. Rules to drive the interpretation system are also derived and added to the system at this time. The internal Prolog form of the production rules are then used in conjunction with the derived rules by the interpreter to build the program structure.

4.5.1 GRAMMAR SYNTAX

1) The top production must be the start symhol.

2) The left hand sides must be a single unique nonterminal. The separator is two dashes followed by a '>', which looks like a leftward pointing arrow.

3) The right-hand-sides consist of a number of different symhols, the terminal symbols appear between single quotes 'IMPLEMENTATION'

4) There are special pretty-printed symbols that indicate newlines, and amount of indentation needed n(3).

5) Angle hrackets around non-terminals <> delimit non-terminals.

6) Semantio actions are delimited by dots .sa(X).

7) The end of produotion is signaled by the slash /

4.5.2 EXAMPLE: GRAMMAR

```
<unit> --> 'MODULE' .id(unit). n(3) .imports.
       n(3) .choose(abstractType,[table,list,stack,tree]),
       n 'END' .retrieve(unit). /

<table> --> <tabletypedefs>
        .choose(tableprocedure,[initialize, sort,
       search, print, insert] )    .more(tableprocedure).
        n /

<tabletypedefs> --> 'TYPE' <range> n(3) ..... /

<range> --> .id(range). '=' '[' .value(lowerbound).
       '..' .value(upperbound). ']' ';' /
```

Figure 4.1: A representative grammar of the start of a table module.

The above example shows a representative sample of the grammar rules needed to create a table implementation module. Decision branches are implemented using the CHOOSE semantic action, that has two arguments, the type, and list of choices. Loops where zero or more repetitions of a production are implemented by the MORE semantic action. The MORE semantic action has an argument that specifies the type of token (production)

the user may want multiples of. All other tokens are taken sequentially in order, with nonterminals <non-term> being attached to their matching production's left-hand side as subtrees, forming a tree structure.

4.5.3 GRAMMAR TO PROLOG TRANSLATION

The translation of the external grammar form to the Prolog form consists of the following algorithm:

```
Get-Right-Hand-Side
  begin
   CYCLE
     read(ch)
     CASE ch OF
       end-produotion       : EXIT
       terminal-delimiter   : transmit unchanged
       non-term delimiter   : make non-term the head
                              of a goal
       semantic-action      : find oorrect special
                              case to set up the cor-
                              rect translation rules.
      END
   END
  end Get-Right-Hand-Side
```

The major routine of the grammar conversion section is called readgram, for read grammar. It has two predicates, that take a single argument, which is a file, as the input routine needs a one character look-ahead to determine which translation routine to use. The first predicate tests the character to see if it is the EndOfFile character, and terminates. The second is a recursive routine that processes a single production each call. The following is illustrates the routine.

```
reedgram(C) :- eof(C).
reedgram(C) :- l-engle(C), getlhs(Lhs,Cout), nl,
          write(Lhs), write('([ '), getrhs(Cout),
          write('#'), write(']).'), skipblanks(C3),
          readgram(C3).
```

The functionality of reed grammar first reeds the left-hend side, which always consists of e single non-terminal, so the GetNonTerminal routine is invoked, end then the errow thet separetes the left-handside from the right hend side of the production is read.

```
/* GET LEFT-HAND-SIDE */
getlhs(Lhs,Ct) :- get0(C), gnonterm(C,List,Cout),
                  name(Lhs,List), reederrow(Ct).
```

The right hand side is much more complicated, beceuse there are meny types of symbols that eppeer in the right hand sides of the productions, terminals, non-terminals, and semantic ections must all be reed and transleted in different ways.

```
/* GET RIGHT-HAND-SIDE */

getrhs(C) :- endproduction(C).

getrhs(C) :- quote(C), getstring(C,String,C1),
             write-token(String), getrhs(C1).

getrhs(C) :- l-angle(C), getnonterm(Nonterm,C1),
             write-token(Nonterm), getrhs(C1).

getrhs(C) :- semantic-ection-delim(C),
             reed(Semaction), write-token(Semaction),
             get0(C1), getrhs(C1).
```

The routine invoked by getrhs to trenslete and write the symbols above is write-token, which merely passes

it's input to the conv-token routine for translation, then when the translated version returns, it handles the output function. The conv-token routine contains the individual rules that specify how individual tokens are to be translated as the grammar is converted into internal Prolog form. The nonterminals are converted as they are read, the terminals are passed unchanged, so that leaves the semantic actions. Each semantic action requires a special rule, as each requires some specific action to be done at interpret time, and this stage must set up the translation action rules.

For example consider the rule to convert a token that matches a grammar symbol that is a semantic action of "id(procedure)". That would mean that the semantic action that should occur at that point in the program is that the user should be prompted to enter an identifier to be bound to a specific procedure. The convert token rule first must generate a unique rule name that can be fired when the time comes to execute the semantic action. The unique symbol is made into the head of a rule, that is given one argument, a variable called Id. The right-hand side of the rule will consist of a call to the routine id_get, with two arguments, Type, and Id, Id will be a variable, and type will be the same value as the argument of the initial id token, in this case 'procedure'. Thus the call to id_get when the semantic action is translated will be

of the form: "getid12(X) :- id_get(procedure,X)". This is a new rule that is placed in the datahase of the Prolog system specifically to execute the semantic action by the call to assert.

```
conv-token(id(Type),Gid) :-
        gensym(getid,G),
        Gid =..[G,Id],  /****** HERE ******/
        assert(Gid :- id_get(Type,Id)).
```

Note the line marked /** HERE **/, there is a strange operator, ' =.. ' which means "make the variable on the left-hand-side become a goal, with the first atom in the right-hand-side list as the head of the goal, and the rest the arguments. This operator allows Prolog programmers to construct goals at runtime, and also to pull them apart.

The behavior of id_get, (see below) is to prompt the user to enter an identifier for the specified type of syntactic symbol, in this case the user will see " procedure Identifier:" The user will type in a value, some syntactic validity checking is done, and the list of declared identifiers are checked. The identifier is then added to the list of identifiers in the Prolog database. This list fulfills the functionality of a symbol table in a conventional translation system.

```
id_get(Type,Id) :- nl,nl,nl,write('   '),write(Type),
                   write(' Identifier: '), read(Id),nl,nl,
                   not id(_,Id), asserta(id(Type,Id)).
```

Another important semantic action is that of retrieving an identifier that has been previously placed in the database. An example of when you would want to do this would be to retrieve the range of an array, or the name of a variable to use in the body of a procedure, where the identifiers were added above in the declaration section. Again a rule must be created at conversion time that will be used at translation time. Since a rule must be created based on input, a call to generate a unique atom to use as the rule head must be generated. Imagine that the retrieve is to get the identifier of an arraytype, so the value of the variable Ty is arraytype. Inside the Prolog database is the following fact: "id(arraytype,NumberArray)." The rule at translation time must return the value "NumberArray", therefore our rule needs an argument to return the identifier, thus the left-hand-side of the rule, the head is the uniquely generated symbol, with an argument, Id, and the right-hand-side is a call to match the Ty value "arraytype", by invoking "id(arraytype,Id)." When this rule is fired at translation time, it matched the Id variable with the fact in the database, and returned the value "NumberArray".

```
conv-token(retrieve(Ty),R) :-
        gensym(ret,Retr),
        R=..[Retr,[Id]],
        assert(R :- id(Ty,Id)).
```

A third very important transletion rule is that needed to set up the functionality of the CHOOSE semantic ection. This is the basic decision branches that the user can take, and results in the different modules that can be created. At translation time, the choose presents a menu of choices that the users heve. Perheps the choice is to choose e data structure to implement, or e choice of a data type, or the choice is a procedure to implement. The grammer gives e list of the choices thet are possible. There are other productions, and a unique production is used based on the choice that the user selects et runtime.

```
choose(Key, Choice, List_of_choices)  :-
    nl, write('  Choose e '), write(Key), nl,
        print_choices(List_of_choices,1,Ct),
        nl,nl,tab(5), reed_num(Ans),
        position(Ans,Ct,List_of_choices,Choice,1).
```

There ere additional sementic actions, but they are similar to the ones shown.

Internal Form of Rules

```
unit(['MODULE', Id1(_), ';' n(3), Imports(_),
   n(3), choose1(_), n, 'END' , retr1(_), '.' ] ).

table([ tabletypedefs(_), choose2(_),
      more(tableprocedure,_), n ]).

tabletypedefs([ range(_), n(3), ..... ]).

range([ Id2(_), '=' , '[' , retrvalue1(_), '..',
     retrvalue2(_), ']' ';' ]).


Id1(X)  :- get-id(unit, X).

Id2(X)  :- get-id(range,X).

choose1(Q) :- choose(abstractType,X,[table, stack,...]),
        P =..[X,Q], call(P).

choose2(Q) :- choose(tableprocedure,X,[Init, sort, ...]),
        P=..[X,Q], call(P).

retr1(X)  :- Id(unit, X).

retr2(X) :- Id(range, X).

retrval1(X)  :-  val(lowerbound, X).

retrval2(X)  :-  val(upperbound, X).
```

Figure 4.2: Prolog facts correspond to each grammar production rule. Each semantic action is converted into a unique procedure call. The rule associated with each semantic action is added to the Prolog database.

4.6 INTERPRETATION

The interpreter calls the Process Command goal passing it one of the following commands. The semantios of eech command will be explored in detail below.

help - a menu of the commands is displayed for the user.

expand - The workhorse of the program. The command 'expands' the program hy huilding e tree, where the root is the start-symbol predicate, and it's lefthend side is a list of arguments. The expand command looks et each element of the list sequentially, huilding the tree hy calling nonterminals and replacing the variehle argument with the list of elements thet constitute the lefthand side of the production.

undo - The undo command traverses the tree, end if it encounters e nonterminal euhtree it queries the user to ask if he ants to delete the suhtree. This can reverse eny of the expanded nodes, end undo eny decisions previously made hy the user, at any level in the tree.

more - The more command expands the nodes that are the semantio action more. here there are possihilities of heving zero or more repetitions of a production, a more node is placed in the grammar. The first time through the expending of a program the more nodes are ignored.

They are only expanded by this command. They can add additional functionality to the module, at the user's request.

save - at the top level of the interpreter the command 'save' will result in a prompt to the user for an filename, and the terminals that make up the program will be written into the file.

quit - first prompts the user for a filename to save program into, exits to the outside system.

4.6.1 CODE FOR INTERPRETATION PROCESSING

```
interp(Pgm,C) :- C==q.
interp(Pgm,C):- process(C,Pgm,Newpgm),
                $$pp(Newpgm),
                get_cmd(NewC),
                !, interp(Newpgm,NewC).
```

The interpreter has two arguments, the program tree structure, and a command, either e)xpand, m)ore, u)ndo, s)ave, h)elp, or q)uit. It then calls process. Process has three arguments, the command, the incoming program data structure, and a new revised program data struoture. The new program is passed to the pretty printer to be displayed on the screen, for the user, and the user is then prompted for another command. The entire routine is then called recursively. The calls terminate when the command is 'q'.

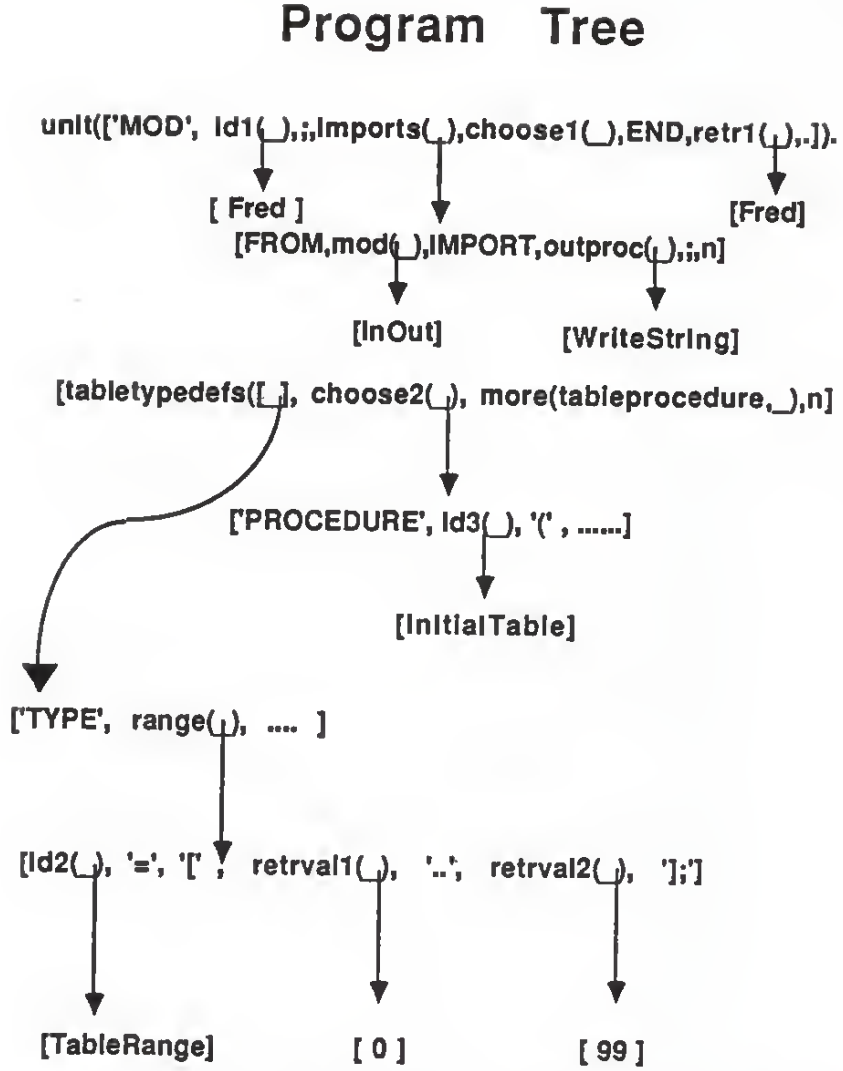

Figure 4.3: The program data structure after being expanded. Each production has a subtree corresponding to the right-hand-side of a production rule. Each semantic action also has values stored as subtrees.

The following is the code for the various processing commands. The explanations for the individual functionality appears below the listing.

THE PROCESS: /* EXPAND */

```
process(e,[],[]).
process(e,[Node|Pgm],[Node|NPgm]) :- atomic(Node),
    !,process(e,Pgm,NPgm).

process(e,[Node|Pgm],[Newnode|Newpgm]) :-
                    Node=..[N,A], isvar(A),
                    !,call(Node), !,
                    paux([Node|Pgm],[Newnode|Newpgm]).

process(e,Pgm,NPgm) :- !, paux(Pgm,NPgm).

paux([N|Pgm],[NN|NPgm]) :- N=..[Nont,Args],
                    !, process(e,Args,Nargs),
                    NN=..[Nont,Nargs], !,
                    process(e,Pgm,NPgm).
```

The first two Process Expand rule matches if the empty list is the argument. This rule serves to terminate processing when the program has been totally expanded. The second occurs when the node is an atom. That means that the node is a terminal and needs to be skipped by Expand. The third rule serves to either expand a nonterminal, or to execute a semantic action. It does this by first retrieving the node, then splitting the node into it's head, and argument. It then tests the argument to see if it is a variable. If it is, then that means it needs expanding, so it "calls" the node, which results in either executing a semantic action, or substituting the left-hand side of the production for the argument. If the argument was not a variable, that means that it had been previously expanded, but the tree needs to be descended and the

nodes of the suhtree need to be expanded, so the final rule calls a helping funotion to pull apart the suh-tree.

```
THE PROCESS : ADD MORE
   /* MORE */
process(m,[],[]).
process(m,[more([T,C])|Pgm],[more([T,C])|NPgm]) :-
        oheck(T), select_choose(T,Ch),
        $$append([Ch],[more([T,X])],C),
        process(m,Pgm,NPgm).

process(m,[Node|Pgm],[NewNode|NPgm]) :-
        Node=..[Nonterm,Args],
        process(m,Args,Newargs),
        NewNode=..[Nonterm,Newargs],
        process(m,Pgm,NPgm).

check(T) :- write('Do You want to add more '),
        write(T), write('s? :(y/n)'),
        read(Response), Response==y.

seleot_choose(Type,Ch) :- ch(Type,List),
        choose(Type,Choice,List),
        Ch=..[Choice,X],call(Ch).

select_choose(Type,X) :-  X=..[Type,Y],
        call(X).
```

The process procedure that processes the MORE command either bottoms out when the program structure is entirely traversed (first rule), it finds a "more" node to match (second rule), or it recursively calls itself until one of the previous two conditions occur (third rule). The action occurs at the second rule. To process a more command, the first aotion is to check with the user if he wants to add a new "type", whatever the first argument is is the "type". More nodes

always correspond with choices, so the choice menu is presented for the user, and the resulting production rule is returned. The rule is then appended to the subtree at the point of the more node, which is retained so that additional procedures can be added at that point.

THE PROCESS : UNDO

```
process(u,[Node|Pgm],[Newnode|NPgm]) :-
                    shrinkit(Node),
                    Node=..[Nont,Arg],
                    Newnode=..[Nont,X],
                    r_scope(Nont), prune(Pgm,NPgm).

prune([],[]).
prune([Head|Pgm],NPgm) :- atomic(Head),
                    !, prune(NPgm).
prune([Head|Pgm], [Newnode|NPgm]) :-
                    Head=..[Nont,Arg],
                    Newnode=..[Nont,X],
                    !, prune(Pgm,NPgm).

shrinkit(Node) :- $$pp([Node]), nl,
                    write('SHRINK THIS NODE? (y/n.)'),
                    read(Response), !, Response==y.
```

The procedure undo takes expanded subtrees, and replaces them with an uninstantiated variable, at the request of the user. It undoes anything that EXPAND does, it even retracts those rules that were asserted in response to semantic actions called by EXPAND as are related to the subtree in question.

4.6.2 RESULT: A MODULA-2 MODULE WRITTEN TO A FILE

```
MODULE Fred ;

  FROM InOut  IMPORT  WriteString;

  TYPE  TableRange = [ 0 .. 99 ] ;

        .
        .
        .
        .

  PROCEDURE InitialTable ( ..............
```

Figure 4.4: When all of the leaf nodes of the tree are terminals, then the system does a tree traversal, and writes the terminals to a file, resulting in a Modula-2 program.

4.7 CONCLUSIONS

This Module Development System generates abstract data types such as the stack shown. The grammars developed so far are capable of generating stacks or queues using either an array or a linked list representation. There are also grammars available for a table, matrix or binary tree. To expand the system, new

grammars must be written. A useful addition to the system would be to automate the translation of a written Modula-2 program into grammar form. This would make it possible for a programmer to introduce working programs into the system without having to hand translate the programs into grammars.

Ideally this system should be embedded in an integrated environment as one of many tools that work together to assist the programmer in system design and implementation.

BIBLIOGRAPHY

[Bar85] Barrett, Kirk R., "A Program Development System using an Attribute Grammar", A Masters Report, Unpublished, Kansas State University, May, 1985.

[Bar79] Barstow, David R., "An Experiment in Knowledge-based Automatic Programming", Artificial Intelligence, Vol. 12, 1979, pp 73-119.

[Bie84] Biermann, Alan W., "Natural Language Programming", in Computer Program Synthesis Methodologies, Reidel Publishing Co., Dordrecht, Holland, ed. Biermann & Guiho, 1984, pp 335 - 368.

[Buh85] Buhr, R.J.A., C.M. Woodside, G.M. Karam, K. Van Der Loo, D.G. Lewis, "Experiments with Prolog Design Descriptions and Tools in CAEDE: an Iconic Design Environment for Multitasking, Systems", in Proceedings 8th Internat'l Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 62 - 67.

[Cam84] Cameron, Robert D., & M. Robert Ito, "Grammar-Based Definition of Metaprogramming Systems", ACM Transactions on Programming Languages and Systems, Jan. 1984, Vol. 6, No. 1, pp. 20 - 54.

[Cla84] Clark, K.L., & F.G. McCabe, "micro-PROLOG:Programming in Logic", Logic Programming Associates, Ltd., Englewood Cliffs, N.J., 1984.

[Clo84] Clocksin, W.F., & C.S. Mellish, Programming in Prolog, second edition, Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984.

[Cop54] Copi, Irving M., SYMBOLIC LOGIC, The MacMillan Co., New York, 1954.

[Cua85] Cuadrado, Clara Y., & John L. Cuadrado, "PROLOG GOES TO WORK", BYTE, August 1985, pp. 151 - 158.

[Dar84] Darlington, J., "The Synthesis of Implementations for Abstract Data Types, A Program Transformation Tactic", in Computer Program Synthesis Methodologies, D. Reidel Publishing Co, Dordrecht, Holland, ed. Biermann & Guiho, 1984, pp 309 - 334.

[Dar84] Darlington, John, "PROGRAM TRANSFORMATION", BYTE, August 1985, pp. 201 - 216.

[Dav85] Davis, Ruth E., "Logic Programming and Prolog: A Tutorial", IEEE Software, September, 1985, pp 53 - 62.

[Fer81] Ferguson, Ron, "PROLOG, A Step Toward the Ultimate Computer Language", BYTE, November, 1981 pp. 384-399.

[Fre85] Frenkel, Karen A., "Toward Automating the Software-Development Cycle", Communications of the ACM Vol. 28, No. 6, June 1985, pp 578-589.

[Fuk85] Fukunaga, Koichi, "PROMPTER: a Knowledge Based Support Tool for Code Understanding", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC 1985, pp. 358-363.

[Gal84] Gallaire, H., "A Study of Prolog", in Computer Program Synthesis Methodologies, Reidel Publishing Co., Dordrecht, Holland, ed. Biermann & Guiho, 1984, pp 173 - 212.

[Gal85] Gallaire, H., "Logic Programming: Further Developments", in IEEE 1985 Symposium on Logic Programming, IEEE Computer Society Press, Washington D.C., 1985, pp 88 - 96.

[Gan85] Ganzinger, Harald & Michael Hanus, "Modular Logic Programming of Compilers", in IEEE Symposium on Logic Programming, IEEE Computer Society Press, Washington D.C., 1985, pp 242 - 253.

[Gog84] Goguen, Joseph A., "Parameterized Programming", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, Sept. 1984.

[Hor85] Horowitz, Ellis, Alfons Kemper, & Balaji Narasimhan, "A Survey of Application Generators", IEEE Software, January 1985, Vol. 2, No. 1, pp 40 - 54.

[Jou84] Jouannaud, J.P., & Y. Kodratoff, "Program Synthesis from Examples of Behavior", in Computer Program Synthesis Methodologies, Reidel Publishing Co., Dordrecht, Holland, ed. Biermann & Guiho, 1984, 213 - 250.

[Kan81] Kant, Elaine & David R. Barstow, "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis" in IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, Sept. 1981.

[Kow79] Kowalski, Robert, Logic for Problem Solving, Elsevier Science Publishing Co, 52 Vanderbilt Ave, New York, 1979.

[Kow85] Kowalski, Robert, "LOGIC PROGRAMMING", BYTE, August 1985, pp. 161 - 177.

[Kuo83] Kuo, J., J. Ramanathan, D. Soni, & M. Suni, "An Adaptable Software Environment to Support Methodologies", in 1983 SoftFair - Software Development: Tools, Techniques, and Alternatives, Computer Society Press, Silver Spring, MD., 1983, pp. 363-374.

[Lee85] Lee, Stanley, "On Executable Models for Rule-Based Prototyping", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 210 - 215.

[Man84] Manna, Z., & R. Waldinger, "Deductive Synthesis of the Unification Algorithm", in Computer Program Synthesis Methodologies, D. Reidel Publishing Co., Dordrecht, Holland, ed. Biermann & Guiho, 1984, pp. 251 - 308.

[Mus85] Musa, John D., "Software Engineering:The Future of a Profession", IEEE Software, Vol. 2, No. 1, January 1985, pp. 55 - 62.

[Ogi85] Ogilvie, John W. L., MODULA-2 Programming, McGraw-Hill Book Company, New York, 1985.

[Par83] Partsch, H. & R. Steinbruggen, "Program Transformation Systems", Computing Surveys, Vol. 15, No. 3, September 1983, pp 199-232.

[Per84] Pereira, Fernando, (ed), "C-Prolog User's Manual, Version 1.4", February 23, 1984.

[Per85] Perlis, Alan J., "Another View of Software", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 395 - 396.

[Pry79] Prywes, N.S., A. Pnueli, & S. Shastry, "Use of a Non Procedural Specification Language and Associated Program Generator in Software Development", ACM Transactions on Programming Languages and Systems, October 1979, Vol. 1, No. 2, pp. 196 - 217.

[Rei85] Reiss, Stephen P., "PECAN: Program Development Systems that Support Multiple Views", IEEE Transactions on Software Engineering, Vol. SE-11 No. 3, March 1985.

[Sha84] Shapiro, Ehud, Algorithmic Program Debugging, The MIT Press, Cambridge, Mass., 1984.

[Shi85] Shimizu, Toru, & Ken Sakamura, " Automatic Tuning of Multi-task Programs for Real-Time Embedded Systems", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 350-357.

[Ste85] Stephens, Mark & Ken Whitehead, "The Analyst - A Workstation for Analysis and Design", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 364 - 369.

[Sus85] Sussman, Gerald Jay, "Intelligent Support for the Engineering of Software" in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp. 397 - 399.

[Tav85] Tavendale, R.D., "A Technique for Prototyping Directly from a Specification", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp. 224 - 229.

[Tei81] Teitelbaum, Tim, Thomas Reps, & Susan Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer", ACM, 1981.

[Vol85] Volpano, Dennis M. & Richard Kieburts, "Software Templates", in Proceedings 8th International Conference on Software Engineering, Computer Society Press, Washington DC, 1985, pp 55 - 60.

[War80] Warren, David H. D., "Logic Programming and Compiler Writing", Software - Practice and Experience, Vol. 10, 1980, pp 97-125.

[Wat82] Waters, Richard C., "The Programmer's Apprentice: Knowledge Based Program Editing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 1, January, 1982.

[Wei84] Weiner, Richard, & Richard Sincovec, "Software Engineering with MODULA-2 and Ada", John Wiley & Sons, Inc., New York, 1984.

[Yeh83] Yeh, Raymond, " Software Engineering ", IEEE Spectrum, November, 1983, pp 91-100.

APPENDIX ONE: SAMPLE TERMINAL SESSION

```
% prolog -L 1000 -G 1000
C-Prolog version 1.4
| ?- [mds].
 ** reading from file **
tempfile reconsulted 2788 bytes 0.466674 sec.
 <unit>

 Legal commands are:

         e.   expand nonterminals
         u.   unexpand a nonterminal
         m.   expand MORE nodes
         s.   save program (so far) in a file
         q.   quit
          r.  to resume


        TERMINATE ALL COMMANDS WITH A PERIOD : r.
 <unit>

Command: e.


   unit identifier: TableMod


   stringtype identifier: str


   lengtb+1 value (integer): 9


  Choose a(n) abstractType

     1) table
     2) sortedlist
     3) queue
     4) stack
     5) tree

     1
 ** reading from file **
tempfile reconsulted 9884 bytes 1.75001 sec.


   range identifier: range
```

```
lbound value (integer): 0

ubound value (integer): 99

arrayty identifier: arraytype

Choose a(n) basety

   1) int
   2) card
   3) real
   4) boolean
   5) char
   6) stringtype

   2

tablety identifier: CardTable

arrayfld identifier: fld

max_index identifier: max

tablePtr identifier: TableType

overflow identifier: overflow

initial-proc identifier: InitialTable

tableparam identifier: T

initialvalue identifier: val

index identifier: index

Choose a(n) tableProcedure

   1) tsort
   2) tsearch
   3) twriteout
   4) tinsert
```

4

procedure identifier: TableInsert

tableparm identifier: T

insvalue identifier: val

```
IMPLEMENTATION MODULE  TableMod ;
  <imports>
  TYPE   str = ARRAY  [ 0  ..  9 ] OF CHAR ;

   range = [  0  ..  99 ] ;
   arraytype = ARRAY  range OF  CARDINAL ;
   CardTable = RECORD
     fld :  arraytype ;
     max :  range ;
  END;

   TableType = POINTER TO  CardTable ;

  VAR  overflow :BOOLEAN;


PROCEDURE  InitialTable ( VAR  T :  TableType ;
                               val :  CARDINAL ) ;
VAR  index :  range ;
BEGIN
    overflow := FALSE;
  NEW(  T );
   T ^.  max :=  0 ;
  FOR  index :=  0 TO  99 DO
     T ^.  fld [  index ] :=  val ;
  END; (* for *)
END  InitialTable ;

PROCEDURE  TableInsert (VAR  T :  TableType ;
                            val :  CARDINAL );
  BEGIN
    IF  overflow THEN
      WriteString(" CAN NOT INSERT - OVERFLOW ");
    ELSE
      WITH  T ^ DO
         fld [  max ] :=  val ;
         max :=  max + 1;
        IF  max >  99
        THEN  overflow := TRUE
        ELSE  overflow := FALSE
      END
    END; (* if *)
  END; (* if *)
```

```
END  TableInsert ;

   <more tableProcedure>

END  TebleMod .

Command: m.
 Do You want to edd more tableProcedures? :(y/n)y.

  Choose e(n) tableProcedure

     1) tsort
     2) tseerch
     3) twriteout
     4) tinsert

     3

   procedure identifier: PrintTable


   tableparm identifier: T


   index identifier: index


Enter e heading for the Table :CerdTable
How many output spaces? (enter a number):8

 IMPLEMENTATION MODULE  TableMod ;

   TYPE   str = ARRAY  [ 0  ..  9 ] OF CHAR ;

    renge = [  0  ..  99 ] ;
    erraytype = ARRAY  renge OF  CARDINAL ;
    CardTable = RECORD
      fld :  erraytype ;
      max :  range ;
   END;

    TableType = POINTER TO  CardTable ;

   VAR  overflow :BOOLEAN;


PROCEDURE  InitialTable ( VAR  T :  TebleType ;
                              val :  CARDINAL ) ;
VAR  index :  range ;
BEGIN
    overflow := FALSE;
   NEW(  T );
    T ^.  max :=  0 ;
```

```
    FOR  index :=  0 TO  99 DO
       T ^.  fld [  index ] :=  val ;
    END; (* for *)
 END  InitialTable ;

 PROCEDURE  TableInsert (VAR  T :  TableType ;
                              val :  CARDINAL );
    BEGIN
      IF  overflow THEN
        WriteString(" CAN NOT INSERT - OVERFLOW ");
      ELSE
        WITH  T ^ DO
           fld [  max ] :=  val ;
           max :=  max + 1;
          IF  max >  99
          THEN  overflow := TRUE
          ELSE  overflow := FALSE
        END
      END; (* if *)
    END; (* if *)
 END  TableInsert ;



 PROCEDURE  PrintTable (VAR  T :  TableType );
    VAR  index :  range ;
    BEGIN
      WriteString("  CardTable "); WriteLn;
      FOR  index :=  0 TO  T ^.  max DO
         WriteCard (  T ^.  fld [  index ]  , 8 );
        WriteLn;
      END (* for *)
    END  PrintTable ; <more tableProcedure>


 END  TableMod .

Command: s.
Type file name to save program in: tablemod
|: .
  IMPLEMENTATION MODULE  TableMod ;
     FROM InOut IMPORT  WriteLn, WriteString ;
  FROM InOut IMPORT  WriteCard ;
  FROM System IMPORT  Allocate, Deallocate ;

    TYPE   str = ARRAY  [ 0  ..  9 ] OF CHAR ;

     range = [  0  ..  99 ] ;
     arraytype = ARRAY  range OF  CARDINAL ;
     CardTable = RECORD
       fld :  arraytype ;
       max :  range ;
    END;
```

```
    TableType = POINTER TO  CardTable ;

   VAR  overflow :BOOLEAN;

PROCEDURE  InitialTable ( VAR  T :  TableType ;
                               val :  CARDINAL ) ;
VAR  index :  range ;
BEGIN
    overflow := FALSE;
   NEW(  T );
    T ^.  max :=  0 ;
   FOR  index :=  0 TO  99 DO
      T ^.  fld [  index ] :=  val ;
   END; (* for *)
END  InitialTable ;


PROCEDURE  TableInsert (VAR  T :  TableType ;
                        val :  CARDINAL );
   BEGIN
     IF  overflow THEN
       WriteString(" CAN NOT INSERT - OVERFLOW ");
     ELSE
       WITH  T ^ DO
          fld [  max ] :=  val ;
          max :=  max + 1;
         IF  max >  99
         THEN  overflow := TRUE
         ELSE  overflow := FALSE
       END
     END; (* if *)
   END; (* if *)
END  TableInsert ;


PROCEDURE  PrintTable (VAR  T :  TableType );
   VAR  index :  range ;
   BEGIN
     WriteString("  CardTable "); WriteLn;
     FOR  index :=  0 TO  T ^.  max DO
        WriteCard (  T ^.  fld [  index ]  , 8 );
       WriteLn;
     END (* for *)
   END  PrintTable ; <more tableProcedure>


END  TableMod .

Command: q.

[ Prolog execution halted ]
```

APPENDIX TWO: SAMPLE GRAMMAR

```
<unit> --> 'IMPLEMENTATION' 'MODULE' .id(unit).
    ';' n(3) .imports(X). n(3) 'TYPE' <string> ';'
    n .choose(abstractType,[table,sortedlist,
    queue, stack,tree]). n 'END' .retrieve(unit).
    '.' /

<int> --> 'INTEGER' /

<real> --> 'REAL' /

<boolean> --> 'BOOLEAN' /

<char> --> 'CHAR' /

<card> --> 'CARDINAL' /

<string> --> .id(stringtype). '='
    'ARRAY' <strlen> 'OF' 'CHAR' /

<stringtype> --> .retrieve(stringtype). /

<strlen> --> '[' 0 <twodots> .val('length+1').
   ']' /

<record> --> .id(record). '=' 'RECORD' <key>
    .more(field). 'END' ';' /

<key> --> .id(keyid). ':'
   .choose(keytype,[card,int,real,boolean,char,
   stringtype]). ';' /

<field> --> .id(field). ':'
   .choose(fieldtype,[card,int,real,boolean,char,
   string]). ';' /

<ptrfld> --> .id(link). ':' .retrieve(pointer).
   ';' /

<pntr> --> .id(pointer). '=' 'POINTER' 'TO'
   .id(node). ';' n(3) .retrieve(node). '='
   'RECORD' n(3) <key> n(3) /

<table> --> <range> n(3) .id(arrayty). '='
   'ARRAY' .retrieve(range). 'OF' .choose
   (basety,[int,card,real,boolean,char,
   stringtype]). ';' n(3) .id(tablety). '='
```

```
    'RECORD' n(5) .id(arrayfld). ':'
    .retrieve(arrayty). ';' n(5) .id(max_index).
    ':' .retrieve(range). ';' n(3) 'END;'  n n(3)
    .id(tablePtr). '=' 'POINTER TO'
    .retrieve(tablety). ';' n n(3) 'VAR'
    .id(overflow). ':BOOLEAN;' n n <tinitialize>
    .choose(tableProcedure, [tsort, tsearch,
    twriteout, tinsert]). n(3)
    .more(tableProcedure). n n /

<tinsert> -->  n n 'PROCEDURE' .id(procedure).
    '(VAR' .id(tableparm).  ':'
    .retrieve(tablePtr). ';' .id(insvalue). ':'
    .retr_asc(basety). ');'     n(3) 'BEGIN' n(5)
    'IF' .retrieve(overflow). 'THEN  WriteString
    (" CAN NOT INSERT - TABLE OVERFLOW ");' n(5)
    'ELSE' n(7)  'WITH' .retrieve(tableparm). '^'
    'DO' n(9) .retrieve(arrayfld). '['
    .retrieve(max_index). ']' ':=' .retrieve(ins
    -value). ';' n(9) .retrieve(max_index). ':='
    .retrieve(max_index). '+ 1;' n(9) 'IF'
    .retrieve(max_index). '>' .retr_val(ubound).
    n(9) 'THEN' .retrieve(overflow). ':=' 'TRUE'
    n(9) 'ELSE' .retrieve(overflow). ':=' 'FALSE'
    n(7) 'END' n(5) 'END; (* if *)' n(3) 'END;'
    n 'END' .retrieve(procedure). ';' n  /

<range> --> n(3) .id(range). '=' '[' .val(lbound).
    <twodots> .val(ubound). ']'  ';' /

<tinitialize> --> n n 'PROCEDURE' .id(initial
    -proc).  '(' 'VAR' .id(tableparam). ':'
    .retrieve(tablePtr).  ';' .id(initialvalue).
    ':' .retr_asc(basety). ')' ';' n  'VAR'
    .id(index). ':' .retrieve(range). ';' n 'BEGIN'
    n(3) .retrieve(overflow).  ':= FALSE;' n(3)
    'NEW(' .retrieve(tableparam). ');' n(3)
    .retrieve(tableparam). '^.' .retrieve(max_index).
    ':=' .retr_val(lbound). ';' n(3) 'FOR'
    .retrieve(index). ':=' .retr_val(lbound). 'TO'
    .retr_val(ubound). 'DO' n(5)
    .retrieve(tableparam). '^.' .retrieve(arrayfld).
    '[' .retrieve(index). ']' ':=' .retrieve(initial
    -value). ';' n(3) 'END;' '(* for *)' n 'END'
    .retrieve(initial-proc). ';' /

<tsort> --> n n 'PROCEDURE' .id(rprocedure). '(VAR'
    .id(tableparm). ':' .retrieve(tablePtr). ';'
    .id(leftindex). ',' .id(rightindex). ':'
    .retrieve(range).  ');' n(3)  'VAR' .id(index1).
    ',' .id(index2). ':' .retrieve(range). ';' n(7)
    .id(temp1). ',' .id(temp2). ':'
    .retr_asc(basety). ';' n(3) <tsortbody> n 'END'
    .retrieve(procedure). ';' n /
```

```
<tsortbody> --> 'BEGIN' n(5) .retrieve(index1).
   ':=' .retrieve(leftindex). ';' n(5)
   .retrieve(index2). ':=' .retrieve(rightindex).
   ';' n(5)  .retrieve(temp1). ':='
   .retrieve(tableparm). '^.' .retrieve(arrayfld).
  '[(' .retrieve(leftindex). '+' .retrieve(right
  -index). ') DIV 2];' n(5) 'REPEAT' n(7) <loops>
  <adjust> n(5)  'UNTIL'  .retrieve(index1). '>'
  .retrieve(index2). ';' n(3) <tests> /

<loops> --> 'WHILE' .retrieve(tableparm). '^.'
   .retrieve(arrayfld). '[' .retrieve(index1). ']'
   '<' .retrieve(temp1).  'DO' n(9)
   .retrieve(index1). ':=' .retrieve(index1). '+1'
   n(7) 'END;' n(7) 'WHILE' .retrieve(temp1). '<'
   .retrieve(tableparm). '^.' .retrieve(arrayfld).
   '[' .retrieve(index2). ']'  'DO' n(9)
   .retrieve(index2). ':=' .retrieve(index2). '-1'
   n(7) 'END;' /

<adjust> --> n(7) 'IF' .retrieve(index1). '<='
   .retrieve(index2). ' THEN' n(9) .retrieve
   (temp2). ':=' .retrieve(tableparm). '^.'
   .retrieve(arrayfld). '[' .retrieve(index1). '];'
   n(9) .retrieve(tableparm). '^.' .retrieve(array
   -fld). '[' .retrieve(index1). ']'  ':='
   .retrieve(tableparm). '^.' .retrieve(arrayfld).
   '[' .retrieve(index2). '];' n(9) .retrieve(tab-
   leparm). '^.' .retrieve(arrayfld). '['
   .retrieve(index2). ']' ':=' .retrieve(temp2).
   ';' n(9) 'INC(' .retrieve(index1). ');' n(9)
   'DEC(' .retrieve(index2). ');' n(7) 'END;' /

<tests> --> 'IF' .retrieve(leftindex). '<'
   .retrieve(index2).  'THEN'  .retrieve(rproc-
   edure). '(' .retrieve(tableparm). ','
   .retrieve(leftindex). ',' .retrieve(index2). ')'
   'END;' n(3) 'IF' .retrieve(index1). '<'
   .retrieve(rightindex). 'THEN' .retrieve(rproc-
   edure). '(' .retrieve(tableparm). ','
   .retrieve(index1). ',' .retrieve(rightindex).
   ')' 'END'  /

<tsearch> --> n n 'PROCEDURE' .id(procedure).
   '(VAR' .id(tableparm). ':' .retrieve(tablePtr).
   ';' .id(searchvalue). ':' .retr_asc(basety).
   ';' 'VAR' .id(found). ':' 'BOOLEAN);' n 'VAR'
   .id(index). ':' .retrieve(range). ';' n(3)
   'BEGIN' n(5) .retrieve(index). ':='
   .retr_val(lbound). ';' n(5) .retrieve(found).
   ':=' 'FALSE;' n(5) 'WHILE' '(NOT'
   .retrieve(found). ')' '&' '(' .retrieve(index).
   '<' .retrieve(tableparm). '^.' .retrieve(max
  _index). ')' 'DO' n(7) 'IF' .retrieve(table-
```

```
    parm). '^.' .retrieve(arrayfld).  '[' .retrieve
    (index). ']' '=' .retrieve(searchvalue). n(7)
    'THEN' .retrieve(found). ':=' 'TRUE;' n(7)
    'ELSE' .retrieve(index). ':=' .retrieve(index).
    '+1;' n(7) 'END; (* if *)' n(5) 'END;' n(3)
    'END' .retrieve(procedure). ';' /

<twriteout> --> n n 'PROCEDURE' .id(procedure).
    '(VAR' .id(tableparm). ':' .retrieve(tablePtr).
    ');'  n(3) 'VAR' .id(index). ':'
    .retrieve(range). ';' n(3) 'BEGIN' n(5)
    'WriteString("'.getheading.'"); WriteLn;' n(5)
    'FOR' .retrieve(index). ':=' .retr_val(lbound).
    'TO' .retrieve(tableparm). '^.' .retrieve(max
   _index).  'DO' n(7) .getimport(basety). '('
    .retrieve(tableparm). '^.' .retrieve(arrayfld).
    '[' .retrieve(index). ']' .howmany. ');'  n(7)
    'WriteLn;' n(5) 'END (* for *)' n(3) 'END'
    .retrieve(procedure). ';'  /
```

APPENDIX THREE: PROGRAM LISTING THE MODULE DEVELOPMENT SYSTEM

```
/*  converts the given grammar, and starts the
    interpreter */
go :-  grammar_convert(grammar), start-symbol(S),
       $$pp(S), interp(S,help), goodbye(S) /* halt */.

/* interprets grammar, replacing non-terminals by their
   right hand side, and executing semantic actions */

interp(Pgm,C) :- C==q, halt.
interp(Pgm,C):- process(C,Pgm,Newpgm),
                $$pp(Newpgm),
                get_cmd(NewC),
                !, interp(Newpgm,NewC).

/* reads commands from users during interpretation */
get_cmd(C) :- nl,nl, write('Command: '), read(C).

/* HELP */
process(help,Pgm,Pgm) :- nl,nl,
   write(' Legal commands are:'), nl,
   nl,tab(8), write(' e.   expand nonterminals '),
   nl,tab(8), write(' u.   unexpand a nonterminal '),
   nl,tab(8), write(' m.   expand MORE nodes '),
   nl,tab(8),
      write(' s.   save program (so far) in a file '),
   nl,tab(8), write(' q.   quit '),
   nl, tab(8), write('  r.  to resume  '), nl,nl,
   nl,tab(8),
      write('TERMINATE ALL COMMANDS WITH A PERIOD : '),
    read(X).

   /* SAVE */
expandimports([],[]).
expandimports([imports(X)|Pgm],[Y,imports(X)|Pgm]) :-
          importmod(Modname,_,_),
          setof(Z,importmod(Modname,N,Z),L),
          R=..[impidents,L],
          Y=..[imp,['FROM',Modname,'IMPORT',R,';',n]],
          retract_all(importmod(Modname,N,_)).


expandimports([n(N)|Pgm],[n(N)|NPgm]) :-
          expandimports(Pgm,NPgm).

expandimports([Node|Pgm],[Node|NPgm]) :-
```

```
          atomic(Node),
          expandimports(Pgm,NPgm).

expandimports([N|Pgm],[New|NPgm]) :-
          N=..[Nonterm,Args],
          expandimports(Args,Newargs),
          New=..[Nonterm,Newargs],
          expandimports(Pgm,NPgm).

process(s,Pgm,Pgm) :-  not importmod(_,_,_), file(Pgm),
   r_scope(unit).
process(s,Pgm,Finalpgm) :- expandimports(Pgm,NPgm),
                           (importmod(_,_,_);
                           NPgm=Finalpgm),
                           process(s,NPgm,Finalpgm).


   /* MORE */
process(m,[],[]).
process(m,[more([T,C])|Pgm],[NewC,more([T,X])|NPgm]) :-
        check(T),
        select_choose(T,C),
        process(e,[C],[NewC]),
        process(m,Pgm,NPgm).

process(m,[more([T,C])|Pgm],[more([T,C])|NPgm]) :-
          process(m,Pgm,NPgm).

process(m,[n(N)|Pgm],[n(N)|NPgm]) :-
          process(m,Pgm,NPgm).

process(m,[Node|Pgm],[Node|NPgm]) :-
          atomic(Node),
          process(m,Pgm,NPgm).

process(m,[Node|Pgm],[NewNode|NPgm]) :-
          Node=..[Nonterm,Args],
          process(m,Args,Newargs),
          NewNode=..[Nonterm,Newargs],
          process(m,Pgm,NPgm).

check(T) :- write(' Do You want to add more '),
            write(T), write('s? :(y/n)'),
            read(Response), Response==y.

select_choose(Type,Ch) :-

ch(Type,List), choose(Type,Choice,List),
          Ch=..[Choice,X],call(Ch).

select_choose(Type,X) :-  X=..[Type,Y], call(X).

   /* EXPAND */
process(e,[],[]).
```

```
process(e,[Node|Pgm],[Node|NPgm]) :-
        atomic(Node),
        !,process(e,Pgm,NPgm).

process(e,[n(N)|Pgm],[n(N)|NPgm]) :-
        !,process(e,Pgm,NPgm).

process(e,[imports(X)|Pgm],[imports(X)|NPgm]) :-
        !,process(e,Pgm,NPgm).

process(e,[more([X,Y])|Pgm],[more([X,Y])|NPgm]) :-
        isvar(Y),
        !,process(e,Pgm,NPgm).

process(e,[more([X,Y])|Pgm],[more([X,NewY])|NPgm]) :-
        process(e,Y,NewY),
        !,process(e,Pgm,NPgm).

process(e,[Node|Pgm],[Newnode|Newpgm]) :-
        Node=..[N,A],
        isvar(A),
        !,call(Node),
        !, paux([Node|Pgm],
        [Newnode|Newpgm]).

process(e,Pgm,NPgm) :-
        !, paux(Pgm,NPgm).

paux([N|Pgm],[NN|NPgm]) :-
        N=..[Nont,Args],
        !, process(e,Args,Nargs),
        NN=..[Nont,Nargs],
        !, process(e,Pgm,NPgm).

/*  UNDO THE EXPANSION OF NONTERMINALS */

process(u,[],[]).
process(u,[Node|Pgm],[Node|NPgm]) :-
        atomic(Node),
        !, process(u,Pgm,NPgm).

process(u,[n(X1)|Pgm],[n(X1)|NPgm]) :-
        !, process(u,Pgm,NPgm).

process(u,[more([X,Y])|Pgm],[more([X,Y])|NPgm]) :-
        isvar(Y),
        !, process(u,Pgm,NPgm).

process(u,[Node|Pgm],[Newnode|Pgm]) :-
      shrinkit(Node),
      Node=..[Nont,Arg],
      Newnode=..[Nont,X],
      r_scope(Nont).
```

```
process(u,[Node|Pgm],[Newnode|NPgm]) :-
       Node=..[N,A],
       !, process(u,A,Newa),
       Newnode=..[N,Newa],
       !, process(u,Pgm,NPgm).

shrinkit(Node) :-
       $$pp([Node]),
       nl, write(' SHRINK THIS NODE??? (y/n.) :'),
       read(Response), !, Response==y.

/* ERROR ROUTINE */
process(C,Pgm,Pgm) :- nl,
        write('Can not perform '), write(C), nl.

/*  GET IDENTIFIER from user at terminal  */
id_get(Type,[Id]) :- nl,nl,nl,
                     write('   '),
                     write(Type),
                     write(' identifier: '),
                     readword(Id),nl,nl,
                     legal-id(Id),
                     asserta(id(Type,Id)).

legal-id(Id) :-
               name(Id,[FirstLetter|Rest]),
               isletter(FirstLetter),
               idsyntax-check(Rest),
               no-dups(Id).

idsyntax-check([]).
idsyntax-check([Letter|Rest]) :-
               isletterordigit(Letter),
               idsyntax-check(Rest).

no-dups(Id) :- not id(_,_).
no-dups(Id) :- not
setof(X,id(X,Id),L).

/*  GET VALUE from user at terminal  */
value_get(Name,[Val]) :- nl,nl,nl,write('   '),
          write(Name), write(' value (integer): '),
          read_num(Val), nl,nl,
          asserta(value(Name,Val)).

/*  save generated source file in a disk file */
file(Prg) :-
     write('Type file name to save program in: '),
     read(Filename), tell(Filename), $$ppsave(Prg),
     told.

/* formatting procedure for printing source program
   into a file, non-terminals are ignored  */
$$ppsave([]).
```

```
$$ppsave(['*'|End]) :-
     $$ppsave(End).
$$ppsave([t(N)|End]) :-
     tab(N), $$ppsave(End).
$$ppsave([n|End]) :-
     nl, $$ppsave(End).
$$ppsave([n(N)|End]) :-
      nl, tab(N), $$ppsave(End).
$$ppsave([more([T,Q])|End]) :-  var(Q),
      $$ppsave(End).
$$ppsave([more([T,Q])|End]) :-  $$ppsave(Q),
   $$ppsave(End).
$$ppsave([Val|Rest])  :-
    atomio(Val), tab(1), write(Val), $$ppsave(Rest).
$$ppsave([Val|End]) :-
    Val=..[Nont,Args], $$ppsave(Args), $$ppsave(End).

/*  formatting procedure for printing program list on
    the screen for the user - non terminals are
    displayed inside brackets     */
$$pp([]).
$$pp(['*'|Rest]):-
     $$pp(Rest).
$$pp([t(N)|End]) :-
     tab(N), $$pp(End).
$$pp([n|End]) :-
     nl, $$pp(End).
$$pp([n(N)|End]) :-
     nl, tab(N), $$pp(End).
$$pp([more([N,Y])|End]) :- var(Y),
     tab(1), write('<'), write('more '), write(N),
     write('>'), $$pp(End).
$$pp([more([N,Y])|End]) :- $$pp(Y), $$pp(End).
$$pp([Val|Rest])  :-
    atomic(Val), tab(1), write(Val), $$pp(Rest).
$$pp([Val|End]) :-
     Val=..[Head,Arg], isvar(Arg),
     tab(1), write('<'),write(Head),write('>'),
     $$pp(End).
$$pp([Val|End]) :-
     Val=..[Head,Arg],
     tab(1), $$pp(Arg),
     $$pp(End).

isvar(X) :- var(X).
isvar([X]) :- var(X).


/*  APPENDS  first two arguments into a list that
    becomes the third argument */
$$append([],X,X).
$$append([X|L1],L2,[X|L3]) :- $$append(L1,L2,L3).
$$append(X,Pgm,[X|Pgm]).
```

```
/* RETRACT SCOPE */
/* retracts identifiers asserted down to a specified
   identifier */
r_scope(Type) :- id(X,Y),((not(X==Type),
   retract(id(X,Y)),r_scope(Type));true).

goodbye(Pgm) :- nl,nl,nl,
             write('*****************************'),nl,nl,
             write('   G O O D B Y E  ! ! !    '),nl,nl,
             write('*****************************').



/*  GRAMMAR CONVERTER */

grammar_convert(Infile) :-
      write(' ** reading from file ** '), nl,
      see(Infile), tell(tempfile),
      getstartsymbol(S,Cin), write(S), write('([').
      getrhs(Cin), write('*'), write(']).'),
      skipblanks(C), readgram(C),seen, told,
      reconsult(tempfile).

getstartsymbol(S,Cout) :- skipblanks(C), l-angle(C),
     getlhs(S,Cout), StSym=..[S,X],
     assert(start-symbol([StSym])).

readgram(C) :- eof(C).
readgram(C) :- l-angle(C), getlhs(Lhs,Cout), nl,
    write(Lhs), write('([').  getrhs(Cout), write('*'),
    write(']).'), skipblanks(C3), readgram(C3).

/* GET LEFT-HAND-SIDE */

getlhs(Lhs,Ct) :- get0(C), gnonterm(C,List,Cout),
    name(Lhs,List), readarrow(Ct).

/* GET RIGHT-HAND-SIDE */
getrhs(C) :- endproduction(C).
getrhs(C) :- endword(C),get0(C1),getrhs(C1).
getrhs(C) :- quote(C), getstring(C,String,C1),
    write-token(String), getrhs(C1).
getrhs(C) :- l-angle(C),  getnonterm(Nonterm,C1),
    write-token(Nonterm), getrhs(C1).
getrhs(C) :- semantic-action-delim(C),
    read(Semaction), write-token(Semaction),
    get0(C1), getrhs(C1).
getrhs(C) :- rword(C,W,C1), write-token(W),
    getrhs(C1).

write-token(X) :- conv-token(X,Out), write(Out),
    write(',').

/* GET NON-TERMINAL */
```

```
getnonterm(Nonterm,Cout) :-  get0(C),
    gnonterm(C,Cs,Cout), name(Nterm,Cs),
    Nonterm=..[Nterm,X].
gnonterm(C,[C|Cs],C2) :-  not r-angle(C),
    get0(C1), gnonterm(C1,Cs,C2).
gnonterm(C,[],Cout) :- get0(Cout).

semantic-action-delim(46).
endproduction(47).
endword(32).
endword(10).
quote(39).

/* GET STRING */
getstring(Cin,String,Cout) :- get0(C),
    gts(C,Cs,Cout), name(String,[Cin|Cs]).
gts(C,[C|Cs],C2) :- not quote(C), get0(C1),
    gts(C1,Cs,C2).
gts(C,[C],Cout) :- get0(Cout).

/*   converts each token of the grammar
       left-hand-side one at a time   */

conv-token(howmany,howmany(X)).

conv-token(getheading,getheading(X)).

conv-token(id(procedure),Gid) :-
        gensym(getid,G), Gid =..[G,Id],
        assert(:-((Gid),
        (id_get(procedure,Id),
        assert(exprt(Id))))).

oonv-token(id(rprocedure),Gid) :-
        gensym(getid,G), Gid =..[G,Id],
        assert(:-((Gid),
        (id_get(procedure,Id),
        assert(exprt(Id))))).

conv-token(id(Type),Gid) :-
        gensym(getid,G), Gid =..[G,Id],
        assert(:-((Gid),
            (id_get(Type,Id)))).

conv-token(val(Name),Gid) :-
        gensym(getval,G), Gid =..[G,Val],
        assert(:-((Gid),
            (value_get(Name,Val)))).

conv-token(choose(abstractType,List),
        Getchoice) :-
        assert(ch(abstractType,List)),
          gensym(choose,Choosecall),
          Getchoice =..[Choosecall,X],
```

```
            assert(:-((Getchoice),
            (choose(abstractType,Choice,List),
            C=..[Choice,X],grammar_convert(Choice),
            call(C),
            assert(assoc(ahstractType,Choice))))).

conv-token(choose(Type,List),Getchoice) :-
            assert(oh(Type,List)),
            gensym(choose,Choosecall),
            Getchoice =..[Choosecall,X],
            assert(:-((Getchoice),
            (choose(Type,Choice,List),
             C=..[Choice,X],call(C),
             C=..[Choice,[Newx|Y]],
             assert(assoc(Type,Newx))))).

conv-token(retrieve(procedure), R) :-
          gensym(ret,Retr), R=..[Retr,[Id]],
          assert(:-((R),
          (id(procedure,Id),
          r_scope(procedure)))).

conv-token(retrieve(initial-proc), R) :-
          gensym(ret,Retr), R=..[Retr,[Id]],
          assert(:-((R),(id(initial-proc,Id),
          r_scope(initial-proc)))).

conv-token(retrieve(rprocedure), R) :-
          gensym(ret,Retr), R=..[Retr,[Id]],
          assert(:-((R),(id(procedure,Id)))).

conv-token(retrieve(Ty), R) :-
          gensym(ret,Retr), R=..[Retr,[Id]],
          assert(:-((R),id(Ty,Id))).

conv-token(retr_asc(Ty),R) :-
          gensym(retasc,Retr), R=..[Retr,[Id]],
          assert(:-((R),assoc(Ty,Id))).

conv-token(retr_val(Nm),R) :-
          gensym(retv,Retr), R=..[Retr,[Id]],
          assert(:-((R),value(Nm,Id))).

conv-token(more(X),more([X,Y])).
conv-token(getimport(B),R) :-
          R=..[gimp,[Impt]],
          assert(:-((R),(assoc(B,Btyp),
          importassoc(Btyp,Impt)))).

conv-token(Tok,Tok).

write_line([]) :- nl.
write_line([H|T]) :- write(H), tab(1),
    write_line(T).
```

```
readarrow(C) :- skipblenks(C0),
        name('-',[C0]), get0(C1),
        name('-',[C1]), get0(C2),
        name('>',[C2]),
        skipblanks(C).

skipblanks(C1) :- get0(C),
        ((not endword(C), C=C1);skipbl(Ct,C1)).
skipbl(C,C1) :- get0(C), not endword(C), C=C1.
skipbl(C,C1) :- skipbl(Ct,C1).

reedword(W) :- skipblenks(C0), rword(C0,W,C2).
rword(C,W,C2) :-
    inword(C), !,
    get0(C1),
    restword(C1,Cs,C2),
    name(W, [C|Cs]).
rword(C,W,C2) :- skipblenks(C0), rword(C0,W,C2).

restword(C,[C|Cs],C2) :-
    inword(C), !,
    get0(C1),
    restword(C1,Cs,C2).
restword(C,[],C).
inword(C) :- not C=46,C>32.
l-angle(60).     /* < */
r-angle(62).     /* > */
l-brecket(123).       /* { */
r-bracket(125).       /* } */
isletterordigit(C) :- isletter(C).
isletterordigit(C) :- isdigit(C).
isletter(C) :- C>96,C<123.
isletter(C) :- C>64,C<91.
isdigit(X) :- X>47,X<58.
eof(26).
newline(10).

read_num(Number) :- get0(N), read_dig(N,Num),
        ((Num==[],!,fail);neme(Number,Num)).
reed_dig(N,Num) :- isdigit(N), get0(N1),
        read_dig(N1,Newlist),
        $$appnum(N,Newlist,Num).
reed_dig(N,[]).

$$appnum(X,[],[X]).
$$appnum(X,L3,[X|L3]).

choose(Key, Choice, List_of_choices)  :-
    nl, write('  Choose a(n) '),
    write(Key), nl,
    print_choices(List_of_choices,1,Ct),
    nl,nl,tab(5), reed_num(Ans),
    position(Ans,Ct,List_of_choices,Choice,1).
```

```
print_choices([],Num,Numout) :-
    Numout is Num - 1.
print_choices([X|Y],Numin,Numout) :-
    New is Numin + 1,
    nl,tab(5), write(Numin),
    write(') '), write(X),
    print_choices(Y,New,Numout).

position(Ans,Ct,Lofc,Choice,Cnum)  :-
       (Ans < 1;Ans>Ct),
       write('  NOT IN LIST OF CHOICES '),
       !, fail.
position(Ans,Ct,[First|Rest],First,Cnum)  :-
       Ans == Cnum.
position(Ans,Ct,[First|Rest],Choice,Cnum)  :-
       New is Cnum + 1,
       position(Ans,Ct,Rest, Choice,New).

id_get(Type,Id) :- nl,nl,nl,write('   '),
      write(Type), write(' Identifier: '),
      read(Id),nl,nl, not id(_,Id),
      asserta(id(Type,Id)).


retract_all(X) :- retract(X), fail.
retract_all(_).

twodots(['..']).

member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).

/* create a new atom starting with a root
   provided and finishing with a unique number */

gensym(Root,Atom) :-
   get_next_number(Root,Num), name(Root,Name1),
   integer_name(Num,Name2),
   $$append(Name1,Name2,Name), name(Atom,Name).

get_next_number(Root,Num)  :-
   retract(current_num(Root,Num1)) , !,
   Num is Num1 + 1,
   asserta(current_num(Root,Num)).
get_next_number(Root,1) :-
   asserta(current_num(Root,1)).

/* convert from an integer to a list
   of characters  */
integer_name(Int,List) :-
      integer_name(Int,[],List).
integer_name(I,Sofar,[C|Sofar]) :-
      I < 10, !, C is I + 48.
integer_name(I,Sofar,List) :-
```

```
   Tophalf is I//10,
   Bothalf is I mod 10,
   C is Bothalf + 48,
   integar_nama(Tophalf,[C|Sofar],List).

importassoc('REAL','WriteReal') :-
    asserta(importmod('RealInOut', 2, 'WriteRaal')).
importassoc('CARDINAL','WriteCard') :-
    asserta(importmod('InOut',2,'WriteCard')).
importassoc('INTEGER','WriteInt') :-
    asserta(importmod('InOut',2,'WriteInt')).
importassoc('CHAR','Writa') :-
    asserta(importmod('InOut',1,'Write')).
importassoc(X,'WritaString').
importmod('InOut',1,'WriteString').
importmod('InOut',1,'WriteLn,').
importmod('System',1,'Deallocate').
importmod('System',1,'Allocate,').

howmany(['',',H]) :- importmod(_,2,_), ask(H).
howmany([*]).
ask(H) :-
   writa(' How many output spaces? (entar a number):'),
   raad_num(H).

getheading([Heading]) :-
    write(' Entar a haading for tha Tahle :'),
    readword(Heading).
getheading([*]).

?- go.
```

A PROLOG PROTOTYPE OF A MODULE DEVELOPMENT SYSTEM

BY

MARITA E. PEAK

B. S., KANSAS STATE UNIVERSITY, 1982

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

MAY 1986

ABSTRACT

This thesis examines several representative automatic program development systems (APDS) and demonstrates an implemention method using a rapid prototype language, .Prolog.

An APDS is a programming environment used to generate correct program modules, freeing the user from typing a program using a text editor. The APDS environment contains knowledge of data structures, algorithms, and target programming language syntax, all of which it uses for program generation.

Represention of program knowledge is integral to the flexibility and usefulness of an APDS. Similarities and limitations of automatic program generation systems are compared and the systems are categorized according to representation schemes.

A module development system was implemented in Prolog. The paper describes the suitability of Prolog both for developing intelligent programming environments, and as a medium for representing program knowledge. A description of the implementation project is included.