1991

A Computerized Formal Methodology For Simulation Software Development.

John H. Barrett
University of Alabama at Birmingham

Order Number 9208068

A computerized formal methodology for simulation software development

Barrett, John H., Ph.D.

University of Alabama at Birmingham, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A COMPUTERIZED FORMAL METHODOLOGY FOR SIMULATION SOFTWARE DEVELOPMENT

by

JOHN H. BARRETT

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer and Information Sciences in the Graduate School, The University of Alabama at Birmingham

BIRMINGHAM, ALABAMA

1991

ABSTRACT OF DISSERTATION
GRADUATE SCHOOL, UNIVERSITY OF ALABAMA AT BIRMINGHAM

Degree Ph.D. Major Subject Computer & Information Sciences

Name of Candidate John H. Barrett

Title A Computerized Formal Methodology For Simulation Software Development

It is widely accepted that the software development process should include formal methodology as well as development tools. This dissertation, concerned with simulation software development, endorses this general view and adds the stipulations that simulation use requires a computerized formal methodology and development tools that are extensible, portable, and easy to use. It is argued, particularly, that:

[1] A sufficient basis for a formal methodology for simulation modeling (continuous, discrete and symbolic) is the methodology developed in this thesis. Rooted in logic programming, the methodology uses both formal semantics and runnable specifications. Demonstrations show how the methodology addresses a proposed core non–numeric simulation language and how systems defined in it are capable of growth.

[2] An implementation tool that satisfies the needs for developing required simulation software is the system presented here. It includes a base processor, an extensible core symbolic simulation language, and associated utilities. Called ASP, for "Augmented Stage2 Processor," it is a major reorganization and updating of William Waite's Stage2 processor, which we subsume along with the latter's potential for Lisp and Snobol features.

The proposed kernel language, called Barrel–F, is provided with a complete formal semantics. Several rule–based processing systems we developed are formalized through runnable specifications. Formal specifications lead us to a discussion of input–output indifference and automatic code generation.

Additional code and systems provide informal software development products and guidelines: [1] utility functions, [2] graphics access, [3] practice and experience for exploiting ASP, [4] linking our new non–numeric capabilities to numeric ones in a GPSS–GASP combined processor, and [5] a new concept of portability, based on the base processor, codes mimicking popular language features, and higher level rule and table processing. Proofs provided in the formal theoretical system put this portability scheme on a completely sound base.

Abstract Approved by: Committee Chairman Kevin D. Reilly

Program Director Warren J. Jones

Date 9/30/91 Dean of Graduate School Cathleen Hamel

ACKNOWLEDGEMENTS

I gratefully acknowledge the guidance and assistance of my advisor, Dr. Kevin Reilly. He has stuck with me and supported me for a long time and I would not have accomplished this without his help. I also appreciate the advice and helpful comments of all my other committee members.

I wish to thank the UAB Graduate School for their financial assistance in the form of fellowships. I also thank my employer, BellSouth Services, for tuition aid.

Most of all I would like to thank my wife and children for their love and support without which I could not have done this. They have endured countless hours of my absence when they have never really understood why.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

APPENDICES

TABLE OF CONTENTS (Continued)

APPENDICES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

CHAPTER 1

INTRODUCTION

This dissertation is concerned with *formal* and *informal* software development with an intended principal application domain of simulation executors and their associated context, i.e., a simulation environment. A formal methodology has been designed and applied to prototype implementations which include a base processor called ASP and an extensible kernel or core language called Barrel-F. The potential of this software to address issues in the application domain as well as the adequacy of the formal methodology to play its assigned role are thereby tested.

The formal methodology is a dual-level logic-based methodology composed of a merged formal semantics system [Moss, 1982] and a runnable specification conceptualization [Davis, 1982; Kowalski, 1979]. Both involve Prolog. The formal methodology has been used to define software tools, to guide in selection of language constructs, and to contribute to automating code development for some software development tasks.

Barrel-F is an extension to a non-numeric software development scheme due to Waite [Waite, 1973] effected through basic extensions of the processor, added embedded code and enhanced utilities. The new base system ASP (or AS2P) stands for Augmented Stage2 Processor. In its modernized and extended form, with added utilities (about a 40% extension in the size of the system), this software is being positioned to serve as a symbolic simulation component integrated with a (numeric) simulation package which entails combined continuous and discrete simulation [Hooper & Reilly, 1983]. This integration

aspect of the research is an extension of earlier work and hence establishes a base for a new paradigm for future simulation software.

The extensibility and flexibility of this new software development scheme includes strong influence from formal theory and methodology. That is, we can exert rigorous control over what software constructs we wish to have and exclude those which have some inappropriate features. The basic processor functions themselves we have already formally defined.

The software is written in the C language and for Unix systems (though we have other operable versions in other languages and under other operating systems). This means we have a degree of portability which provides the capability to move the software to several machine types such as those available in our university and other organizations, e.g., Sun, Sequent, Cray.

We do not adopt the view that all software within and generated by this system *must* be developed from a formal point of view. The extensibility and flexibility of the development facility provides many aids for experimental software development and prototyping. We have experimented in this mode on many occasions and results of this mode appear in later chapters and the appendices. We call these experimental products, Barrel–E, (an extension beyond the formal system, Barrel–F). Barrel–E and Barrel–F, sometimes referred to collectively as Barrel, can be viewed as a family of related language features. Organization within this family will be discussed later.

The importance of (the formally defined) Barrel–F as a core upon which to build is reflected in our providing a formal definition for it, along with a complete implementation. The major objective is to provide individual modelers or groups of modelers with a core which can be systematically extended. Since extensions should be accompanied by an associated formal reasoning system, development

tools are needed. A framework for the establishment of such support tools is developed as well as examples and demonstration prototypes. Similarly, an approach is also presented for dealing with the issue of automating the relations between the reasoning system and the products developed within it. We consider this capability a sine qua non for applications in complex simulation environments. We address this topic by providing examples of how we have "hand-crafted" some of our own extensions to the original Barrel-F definition *and* how we can automate portions of this process.

Both the developed software and the theoretical methods are assessed in relation to a new paradigm for simulation software development within an automated simulation environment. The proposed new paradigm includes formal methods as an aid to the specification, design and implementation of model components, as well as components of the simulation environment itself.

This research has also led to an enhanced notion of portability. A three-level portability scheme is proposed, developed and demonstrated. The scheme entails these elements: 1) an abstract machine implementation scheme; 2) language construct mimicking capability within a pattern directed capability offered by the system to users; 3) prototypical table processing methods which, in demonstrations, connect as many as nine different processors on five different machines.

We are now ready to state our main point, our thesis.

1.1 THESIS

Software for simulation systems should be developed within a framework which has both a strong theoretical foundation and a useful and practical application package. Furthermore, to be truly relevant to the needs of the users, a

computerized formal methodology is preferred over a manual methodology, and the implementation tool must be extensible, portable, and easy to use.

A sufficient basis for a formal methodology for simulation modeling in a broadened sense of combined continuous, discrete and symbolic simulation is a methodology we have developed in logic programming; an implementation tool that satisfies the stated needs for developing software for simulation models and environments, with primary emphasis on the symbolic and non–numeric areas, is the system we have developed, which includes a base processor, a core facility forming an incipient symbolic simulation language, and much associated utility code.

Note that this *thesis* entails more than adapting formal methods. It includes concepts of practical methodology as well as implementation tools. The attributes of the implementation tool, therefore, are regarded as *fundamental to the thesis*.

Broken down to simplest units, we see in the thesis needs for discussions centering on these principal themes:

- non–numeric simulation and simulation environments
- computerized formal methodologies
- software development tools

These discussions, which constitute sections 2–4 of this chapter, will reveal a comprehensive methodology for developing simulation (and other systems) software. Formalization is a cornerstone and a substantial amount of experience dealing with the pragmatic issues will be on display and play an effective complementary role.

Within each of these thematic pursuits, we need to identify and develop tools suitable for the scheme of software development. We also need to back up our theoretical ideas by pragmatic decisions and prototypes to provide a complete

overview of how, in our purview, symbolic simulation software should be developed.

1.2 NON–NUMERIC SIMULATION AND SIMULATION ENVIRONMENTS

Simulation provides a major motivating factor in our research. The current state of the art in general purpose simulation systems incorporates combined continuous and discrete simulation, represented by commercial products and prototype processor packages such as GGC (GPSS–Gasp Combined) [Hooper, 1983]. A growing importance of symbolic (AI, non–numeric) elements in simulation and of symbolic simulation itself is seen in use of production systems in simulation environments and models. We have concluded that exploiting the full range of non–numeric processing, in conjunction with full numeric capabilities, is moving the art to a higher plane [Reilly, Barrett & Lilly, 1987]. We wish to provide systems which offer, on behalf of this potentially emerging state of the art, "combined continuous–discrete–symbolic simulation" capabilities.

A prototype for this purpose has been designed, with main elements being UAB's own GGC and ASP systems, the latter of which is discussed at length in this dissertation. ASP provides primary support for non–numeric processing and GGC provides primary support for numeric processing. The two can be made to work well together, and some of our remarks mention the prototype where we linked these two systems [Barrett & Reilly, 1987].

The principal areas of non–numeric processing we focus on in this dissertation are:

- rule based and table processing
- Lisp, Snobol and Prolog like features
- other conventional programming language features

- extensibility, flexibility and portability
- formality

Let us briefly comment on each of these.

1.2.1 Rule Based and Table Processing

Rule processing "within the Barrel" is provided through a combination of tools and features centering on decision table processing, a derivative expert system capability, and openings to other forms of table processing such as relational databases. We adopt the point of view of Weiss and Kulikowski, also adopted in the work of our UAB predecessor, A. Salah, that table processing, specifically decision tables, provides a means of representing "extended production rules" [Weiss & Kulikowski, 1984]. Salah's work on decision tables in logic programming showed important relationships to relational databases [Salah, 1986; Reilly, Salah & Yang, 1987]. While it is clearly outside the scope of this thesis to develop the potential of combining all of Salah's ideas with those presented here, we would be remiss if we did not mention the existence of this potential.

Formalism can be based on adopting Marvin Minsky's statement that production systems are "just" another way to program a system or others' claims that, in so–called table–dominant systems, decision tables can be used to replace programming altogether (see [Metzner & Barnes, 1977] for this and related points). In this purview we are engaged in viewing tables in programming language terms. This legitimate view means using the Moss–based formalism (see below).

Alternatively, we can raise the level of view to the programming systems (or sub–systems) level, a purview which reflects the role production systems may play in a simulation environment context. Then, logic programming specifications is the called–for formal approach (see below). This aids in, e.g., following Salah, Reilly and Yang into setting up connections between decision table processors, relational

databases and other systems. The nine–processors and five–machines portability scheme is consonant with this view, of course.

We have incorporated some rule based processing features in Barrel–F, as described in the first section of Chapter 2. Several additional ones appear in the collection of experimental codes in "extended Barrel", Barrel–E (Chapter 10). Barrel–F, of course, is that formally defined portion in the Barrel family which we designate as a "core" or "kernel" facility, a minimal recommended non–numeric simulation processing capability, a base upon which to build. Thus, some processing of tables is formally defined in Barrel–F's formal semantics, while other features of it are handled formally through the runnable specifications portion of our merged formal system. Closely related to this portion of the formal system are some efforts in producing specifications automatically from informal dialog programs. A method of automatically providing formal specifications for table processing systems is covered in the first section of Chapter 3.

In Chapter 4 (section 3), a discussion on portability is directed toward facilities provided through several table processors available to us. Some of these processors are self–contained systems. We established contact with such systems by taking output from our table–creating system (see below) and producing input files for these systems, e.g., through table reformatting programs and code augmentation schemes.

In Chapter 10, we introduce another component of an overall table processing system, the "presentation processor." We discuss and demonstrate some of the main capabilities in table presentation developed through use of ASP mechanisms.

Finally, Chapters 11 and 12 describe a system, again developed within ASP, for *creating* decision tables from scratch and (re–)formatting them for further processing by several table processors, some of which were developed earlier by us.

It is this (creation) system that starts up the chain of activities in the decision table system processing enterprise. It produces output files which are then reformatted and augmented in various ways, on their way to other table processors. The portability afforded by this scheme is reflected in the numbers of processors reached as a result of these maneuvers: an impressive number of nine different systems in five different programming languages is entailed.

1.2.2 Lisp, Snobol, and Prolog Like Features

Lisp–, Snobol– and Prolog–like mechanisms are deemed as virtual necessities for manipulating lists, strings and relations. The ability to do recursive programming and the use of pattern matching capabilities in the ASP base are important ingredients in connection with these features.

Much of our work reflects our interest in Lisp, Snobol and Prolog in the form of the language constructs we have chosen to implement, the syntax we prefer, and our reliance on pattern matching and recursion. More specifically, in the third section of Chapter 3, we discuss a table processor implemented in a style very similar to its Prolog specifications. In Chapter 9 we focus on components of Barrel–E called kits – groups of related programming language constructs. One of these kits is modelled after features of Lisp. Another is modelled after features of Icon, a derivative of Snobol. Chapter 9 (second section) also discusses communications between ASP and a purely functional version of Lisp, Lispkit Lisp.

1.2.3 Other Conventional Programming Language Features

In addition to these mechanisms, other more conventional types of programming language features are deemed to be needed, for example, to facilitate porting of code from popular languages like C, Ada and Pascal, and to interface smoothly with GGC, which is written in C and/or Fortran. Then, too, simulation

applications dictate that we be able to handle the kinds of code which show up in typical "event" routines (in GGC and other processors, also); such code is often elaborated as procedural code. Barrel–F contains some of these types of features, but Chapters 8, 9 and 10 focus on many other features which more fully reflect our effort to deal with these types of programming language features.

Those portions of Barrel which constitute *explorations* of language features are in Barrel–E, defined earlier. We do not provide formal definition for these features, since to do so is outside the scheme: the primary purpose is that we provide definitions only for a core or kernel; users, then, are free to add more code, backing it up with experiments as suits their needs. The philosophy is that *we* purposefully avoid a large facility at this time, seeking rather a core from which informal exploration and development can ensue, and then ultimately made formal.

1.2.4 Extensibility, Flexibility and Portability

We need extensibility because we believe it is best to treat symbolic simulation as a growing and changing area of computer science. We seek room for individual options. It seems prudent, therefore, not to make hard guesses about the future, which might turn up later to be restrictive in some way.

We need flexibility to do things, e.g., like make an eclectic choice based on use of (usually a few) Lisp–, Snobol– and Prolog–like features and the more common code capabilities working in collusion. Flexibility is needed also to port software to new architectures as the need arises. Extensibility and flexibility, therefore, permeate this research work.

In the second section of Chapter 2 we show how our formal methodologies can be *extended* as our languages are extended. In Chapter 4 we show *flexibility* through three instances of *portability* which can be used together or separately. Our

descriptions of the ASP itself show its extensibility and flexibility. Our table processing system, described in Chapters 11 and 12, provides the flexibility of creating tables to be processed by several different processors.

1.2.5 Formality

Finally, we need formality to confer on our work a good, firm computer science basis. We have already introduced our formal methodologies and they are described fully in Chapters 2 and 3.

1.3 FORMALISM AND LOGIC

From some of our earlier statements we may perceive that there is another group of ideas that are only now beginning to have a significant bearing on software development in relation to simulation modeling. This collection of ideas is that of using formalized methodologies which allow us to reason about the models as we build (and use) them. We view some of this dissertation's arguments and demonstrations on introducing formal methodology as a major contribution, in principle and in practice, to the development of sophisticated B–Units (Builder Units), in conjunction primarily with the K–Unit (Knowledge Unit), in a conceptualization of a simulation environment known as the BEAK [Reilly & Dey, 1987; Reilly, Jones, Barrett, Salah, Strand, Autry & Rowe, 1984; Reilly, Jones & Dey, 1985; Reilly, Jones, Lyons, Payer, Ramer & Dey, 1985; Reilly, Ramer, Dey, Suter, Lyons & Byoun, 1986]. (BEAK is a representation of simulation environments as four primary units – Builder, Executor, Analysis and Knowledge; we discuss our relationship to the BEAK in detail in section 6 of Chapter 14.)

1.3.1 Choice of Formal Tools

One of our major objectives from the outset, has been to identify appropriate formal tools and associated programming language and system theory that would

support our simulation environment conceptualizations. The methods had to be sufficiently adaptable to change. In particular, we expect that non-numeric computing, both in the E-Unit (Execute Unit) and in the other simulation environment units, is only partially understood at this point in time. If we are correct, a tool that is unable to respond to change would be nearly useless.

Additionally, the appropriate tool must have an implementation that is flexible and not burdensome when moved to new machines. It should be robust even in the presence of extensions to machine contexts that entail radical changes in methodologies or in other cases where the issue is one of harnessing an incredible amount of computing power. (We are thinking of distributed systems and networks of supercomputers.)

Based on the foregoing reasoning we chose to base our formalizations on (formal) logic. Our point of view, accordingly, is one of primacy of logic over other formal systems. Historically, this links us with the ideas evolved over the period of the great works by Boole, Frege and Whitehead and Russell. Translated into contemporary terms, this is viewed tantamount to some form of logic programming.

Moreover, our intent is that we stay as close to Horn Clause forms as possible, so that we can make Prolog our workhorse. What Kowalski [1979] calls metalanguage processing is an acceptable enhancement of Prolog, when the metalanguage is programmed in Prolog and some of our detailed level work might be cast in this context. We don't, however, make this a major point in our work at this time. Recall that our primary stipulation is that we *require* our formal tool to be programmable and executable. Very interesting is the fact that some software we developed in ASP has already been used to emulate certain core features of Prolog;

it could probably be used as a vehicle to implement Prolog itself, though clearly this would be outside the scope of this dissertation.

A primary thrust of this dissertation is that we explore methodologies for developing and utilizing tools that promote *combining* non–numeric *with* numeric activities in simulation environments. We show how formal tools (we develop) are utilized in a framework which combines a formal semantics approach and a runnable specification notion, and in which the nature of the code is immaterial (i.e., any combination of numeric and non–numeric).

1.3.2 Programming Language Definitions

In the case of formal semantics there are a multitude of options available to the computer scientist, and references are provided for several of these within the dissertation. We adopt a particular stance in relation to the underlying definitional framework, the most important two features being: first, the constraint of logic, that is, that the base methodology be a well–understood formal (mathematical) logic system in which we present our definitions; and, second, that it be programmable on a computer.

We found that some necessary key features exist in a form of relational semantics whose recent history is connected principally with the development of metamorphosis grammars [Colmerauer, 1978] and the logic programming (and Prolog) movement [Kowalski, 1979]. We demonstrate the value of techniques emerging from these activities to software operating *at different levels* in the programming hierarchy, i.e., low–, intermediate– and high–level constructions.

More specifically, we aim to exploit to the maximum extent possible the opportunity provided by having a formal logic tool implemented within a programming language. We use Prolog to help us formally develop another tool – a symbolic software development tool (ASP) together with a system of

programming language features and programming systems (Barrel – specifically, the Barrel–F portion). We use the formal tool to help us accomplish the following tasks which will be discussed later in this document:

- design and implement a programming language,
- perform formal analysis before implementation,
- formally analyze implementation tools,
- aid in extending formal definitions,
- develop programmable formal definitions,
- provide one formalism for all aspects of the definition,
- provide a formal basis for the core of an extensible language

1.3.3 Programming Systems Definitions

"Programming systems" is a phrase we use to denote entities consisting of the highest level code groups we deal with in this dissertation. It might be easiest to characterize them negatively, i.e., they are not assembly level; they are not high level in the sense of languages such as C, Fortran, Ada and Pascal. On the positive side of the ledger these entities are often very high level commands in some cases, in analogy to 4th generation systems. A second case consists of utility processes and routines called up specifically through some code word, as e.g., in some features utilizing graphics included in our system.

For these systems we adopt the same theme as with programming languages: primacy of formal logic methodology. The larger context of (programming) systems, as is the case with programming language constructs, can be nicely described in a lifecycle diagram such as that depicted in Figure 2.1. In fact, some of the very high level instructions can be handled as if they were just another high–level construction. In other cases, we give our systems' specifications (directly) in formal logic. Consistent with our earlier position, they must be developed in a

form of logic which has a programming basis. So, again, we call on Prolog, a programming language based on first order logic, this time in the role of *specification language*.

Prolog, as a specification language, has several advantages over many other formalisms. Since Prolog is executable on a computer, it allows a more detailed analysis of the specifications at the earliest possible stage in the software lifecycle. The specification can be analyzed and executed before the implementation takes place. This helps pinpoint problems with the specifications without the expense of an incorrect implementation. Such specifications can serve as a prototype, or, in many cases, may serve as an adequate implementation, eliminating the design and implementation phases from the lifecycle, and (trivially) solving the correctness problem in the wake.

Prolog has a firm theoretical foundation with well-defined fixpoint or denotational semantics as well as its proof theoretic semantics [Moss, 1981]. For us the advantage extends to the fact that we use Prolog in other complementary studies, i.e., it has in effect been the method of choice for us. The decision to use Prolog connects our work to prior work, e.g., that of Ruth Davis in her paper on "runnable specifications" [Davis, 1982]. Kowalski also makes a strong case for the use of Prolog as "runnable specifications" [Kowalski, 1979, 1984a, 1984b]. For illustrative purposes, we use logic programming to create a runnable specification for an implementation of a decision table presentation processor to be implemented within ASP as part of the Barrel family.

1.4 SYMBOLIC SOFTWARE DEVELOPMENT TOOLS

Assuming methodologies are in place, encompassing the formal aspects of developing symbolic simulation software as we have depicted them above, we are

ready to turn our attention to the tools which will promote implementations that are faithful to the formal specifications.

We have developed such a tool, which we submit as a candidate for providing the capabilities conducive to faithful implementations. The tool is called ASP (or AS2P, more recently). It is described thoroughly in Chapters 6, 7 and 8 of this dissertation.

ASP is an extended version of a widely acclaimed general purpose macro processor (Stage2). Significant changes have been made to Stage2 to produce the current version of ASP. New features have been added and its method of processing has been altered slightly. Relatively small changes in some cases have brought about a substantial difference in the characteristics of the processor.

Stage2 is described as a processor which accepts character strings as input and transforms them according to a set of definitions provided by the programmer [Waite, 1973]. A typical example might be transformation from one programming language to another. ASP retains the string transformation capabilities of its predecessor, but it performs a different function as well. ASP can interpret the strings as programming language statements and execute them. This change in orientation induces a powerful and flexible tool for experimenting with and developing programming language features and software systems.

We show ASP's usefulness in developing each of the capabilities we earlier deemed most useful to us for symbolic simulation, that is: 1) rule based (table) processing; 2) Lisp, Snobol and Prolog like features; 3) other more conventional programming language features; 4) extensibility and flexibility; 5) formality.

These capabilities are demonstrated in the context of practical applications and in the form of table processing systems and programming language features. The latter are or can be incorporated in a series of "kits" of programming language

features, languages and systems tailored toward specific needs and programming systems, often table processors, all implemented through the use of ASP. We discuss these in detail in Chapters 8 through 12 of this dissertation.

We have used the same technique for defining Barrel–F to provide a formal definition of a portion of the processing that takes place within ASP. It allows us to have a programming language developed inside of a formally defined implementation tool, the latter itself being viewed in its programming language frame (it being in effect the assembly language of an abstract machine). The formal definitions produce a formal description of one abstract machine, viewed through its machine language, in terms of another abstract machine's native language. We take a closer look at this later in this document.

CHAPTER 2

COMPATIBLE EXTENSIBILITY IN DEFINITIONS AND SYSTEMS

The major "theme" of this dissertation is to develop and/or utilize certain tools in a variety of areas of computer science, e.g., tools that help to combine numeric and non–numeric activities in simulation environments. We also show that the tools can be developed in a theoretical framework which combines a formal semantics approach and a runnable specification notion. In this chapter, we focus on the formal semantics approach, or, more precisely, formal definitions of programming languages.

In this case there are a multitude of options available to the computer scientist, and references are provided for several of these. We present a particular point of view in relation to the underlying definitional framework, the most important two features being that it be a well–understood formal (mathematical) logic system in which we present our definitions; and that it should be programmable on a computer, not only because this makes it easier to check the definitions but more importantly because the programming of interest to us involves extensible systems.

Our point of view is one of primacy of logic over other formal systems. Historically, this links us with the point of view evolved over the period of the great works by Boole, Frege and Whitehead and Russell. We found that the main features we need exist in a form of relational semantics, which has a more recent history connected principally with the development of metamorphosis grammars [Colmerauer, 1978] and the logic programming and Prolog movement [Kowalski,

1979]. We will demonstrate the value of techniques emerging from these activities to rather complex software operating at different levels in the programming hierarchy.

Specifically, we want to exploit the opportunity provided by having a formal logic tool implemented as a programming language, Prolog. We use Prolog to help us formally develop another tool – a programming and systems development tool – called ASP. We use the formal tool to help us accomplish the following tasks:

- design and implement a programming language,
- do formal analysis before implementation,
- formally analyze the implementation tools,
- easily extend formal definitions,
- develop programmable formal definitions,
- provide one formalism for all aspects of the definition,
- provide a formal basis for the core of an extensible language.

We now discuss each of these tasks in more detail before outlining the methodology used to accomplish them.

- design and implement a programming language

The advantages of formally defining a programming language are well documented in the literature [Burstall, 1969; Cleaveland & Uzgalis, 1977; Dijkstra, 1976; Hoare & Lauer, 1974; Marcotty, Ledgard, & Bachman, 1976; Neuhold, 1978; Pagan, 1981; Rustin, 1972]. Typical of these advantages [Pagan, 1981] are:

- standardization of the programming language,
- reference for users,
- reference for implementors,
- proofs about programs,
- proofs of implementations,

- automatic implementations,
- improved language design.

Virtually all these advantages appear in our work and appropriate reference is made as needed. In addition, as we shall see below, other advantages are realized in our particular realm of operation.

We have designed a programming language called Barrel–F. Part of the design process included a formal definition for the language. We have also implemented the language through the facilities of the ASP system. The particular details will be discussed later.

The formal definition had a definite impact on the design of the language. Several improvements were made to the design because of issues brought out by the definition. For example, the assignment statement was modified to allow the evaluation of arithmetic expressions because, in developing the formal definition, we realized that it had been omitted from the design. The if/then statement dramatically changed in style and function. The type of looping control structure to be incorporated in the language was decided upon.

The formal definition also had a definite impact on the implementation of the language. The formal definition was used to guide the implementation in several ways. We list some of the more influential features of the definition in this regard and then discuss them more fully below.

- formal analysis before implementation
- formal analysis of the implementation tools
- easily extended formal definition
- programmable formal definition

It should be noted that these features are also included among the tasks listed above.

- do formal analysis before implementation

Most of the advantages of formal definitions experience a boost in effectiveness if formal analysis is strategically placed within the lifecycle. (We provide Figure 2.1 to define the lifecycle notions we need; this description, of course, is quite standard.)

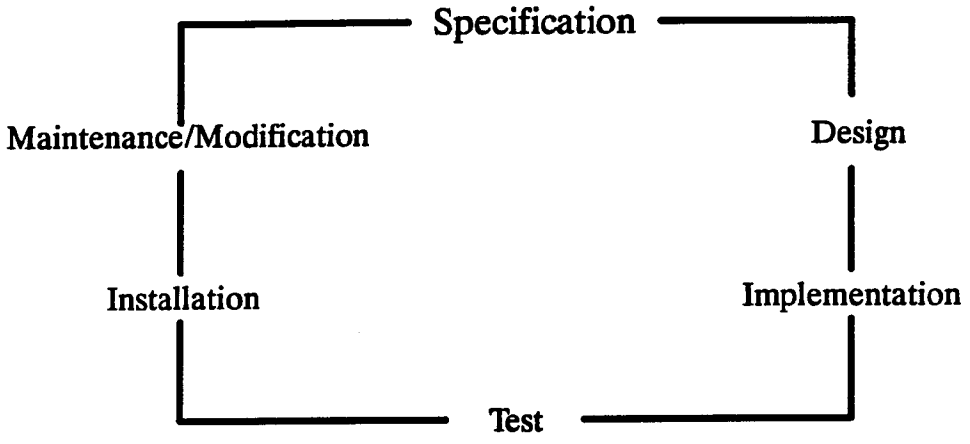


Figure 2.1. The Software Lifecycle.

For example, for the usual case of a one-time definition of a language, the formal definition should precede language implementation, consistent in time sequence with the lifecycle diagram of Figure 2.1. In practice, formal analysis of a programming language is most often done after the language has been widely used for some time. Thus, improvements in the language design brought out by the formal analysis are not likely to be incorporated in the language (or at least in the early versions).

Jean Ichbiah, the principle designer of Ada, states:

> Ada represents the first instance in the history of programming languages of things being done in the right order. .... For Ada, we first created the design, ... [then] we made sure the description of the language was precise, ... then a validation facility was produced, ... finally, the compilers appeared. [Ada, 1984]

Barrel–F was developed along similar lines in that formal analysis was done before the implementation. Also, a validation facility (in the form of test programs, see Section 2.4) was developed before the implementation. This facility was used to validate both the definition and the implementation.

- formally analyze the implementation tools

Not only have we defined the language but we have also defined the tool used to implement it. We feel this enhances the probability that the implementation will be correct. In our case, the implementation tool is ASP. We used the same methodology to define ASP as we used to define the language implemented through ASP. In many instances, the definitions overlap because most of the language being implemented can be used to implement the language. A full description of the definition can be found in Section 2.2 while the full description of ASP is found later, in Chapters 5, 6 and 7.

- easily extend formal definitions

Barrel–F is a highly extensible language. Many extensions have already been developed and can be used in any combination with Barrel–F (or even without it; e.g., see Chapter 9). When a language that has been formally defined is extended then the formal definition may become obsolete (though not always if the change is an "extension" rather than a "modification"). We believe that, when possible, the definition should be extended along with the language. Whether the extensions are formally defined or not depends a great deal on whether the definition can be easily extended. We intend to show that the definition of Barrel–F is easily extended (see Section 2.2.1).

- develop programmable formal definitions

The definition should be programmable on a computer. This may place an extra burden for some theoretical purposes, but when practical goals exist it is

extremely important. For example, in our case, our defined language is extensible and the computerized definition helps in containing bookkeeping overhead.

The methodology we have chosen for the formal definition can also serve as a computer program. Thus, the definition also serves as an implementation of the language. However, we have chosen other avenues of implementation as well since an executable definition does not always yield a user friendly or efficient implementation. We have also tried to make the implementation available on as many machines as possible by using several portability techniques (see Chapters 4 and 7).

Formal definitions can be used to answer questions that programmers or compiler writers may have about the implementation, syntax, or semantics of a programming language. Questions such as "does this statement work with this data type" or "how does this statement work inside a loop" frequently arise. Trying to answer the questions by looking at the definition alone can be very tedious and complicated and even misleading, depending on the complexity of the definition. When a definition is put on a computer and executed it allows questionable statements to be used as input to the definition. Examination of the results along with, perhaps, examination of the definition should answer most of the questions.

Execution of the definition also allows testing for completeness and accuracy. Programs which thoroughly test the definition should be developed. These same programs can be used to test the implementation of the language (i.e., the compiler or interpreter).

- provide one formalism for all aspects of the definition

Most of the time, the methodology used to define the syntax of a language is not used to define the semantics. Indeed, most formal methodologies are suitable for defining only one aspect of the language, either the syntax (e.g., Backus–Naur

Form) or the semantics (e.g., axiomatic semantics). Thus, most of the time, two or more separate methodologies are used to provide a complete definition. We desire a methodology that can be used for both the syntax and the semantics.

- provide a formal basis for the core of an extensible language

C.A.R. Hoare, in his 1980 Turing Award Lecture, recommends starting with a small language and for certain applications, making specialized extensions of the language toward these applications [Hoare, 1981]. Not only is Barrel–F a small language but it is extensible as well.

We believe the small language should be formally defined to provide a stable environment from which to build. Many times in an extensible environment programmers must contend with descriptions of features that range from non–existent to very detailed. It becomes very difficult to keep up with what others have done. One way to help alleviate the problem is to provide a core set of language features that can be used by all programmers. The core should remain fairly constant. It should contain those types of statements which are used most often by the programmers. That way the most popular statements are well defined and used by everyone. We view Barrel–F as that core for the Barrel/ASP programming environment. Formally defining the core stabilizes the environment even further.

2.1 DESIGN OF BARREL–F

During the development of software (i.e. programming languages and systems) for the Barrel/ASP system, the need for a core set of language features emerged. Barrel–F is that core and is directly portable to all implementations of Barrel/ASP. It provides a stable base from which to begin software development in Barrel/ASP so that programmers do not have to reimplement these language features. Thus, all

programmers can have a common set of language features which they can extend to their particular application.

Experience with the Barrel/ASP system was the most influential factor in designing the Barrel–F language. Those features that were considered necessary and useful no matter what the application were included.

Most of the features of Barrel–F have counterparts in high–level languages such as C, Ada, Pascal, and Basic. Also included is a very high–level feature which relates to a particular application (table processing) around which most of the software development projects of Barrel/ASP have centered (see Chapter 11).

Some of the features are briefly and informally described in Figure 2.2 (more detail on all of the statements is given in Appendix 2.1). The language consists of 5 assignment statements (4 arithmetic in nature), 4 input/output statements, 1 very high–level statement which operates on tables, 4 stack manipulation statements, 6 queue manipulation statements, 4 control statements, an execute statement which can execute data, 9 functions including 3 which operate on stacks and 4 which operate on queues, and a data structuring tool which operates on sparse arrays with integer and character string indices similar to the MUMPS programming language [Walters, Bowie & Wilcox, 1982].

2.2 FORMAL DEFINITION OF BARREL–F

For reasons discussed above we feel the design of our programming language would not be complete without providing a formal definition. We seek a formal methodology which most closely relates to those goals and tasks outlined in the introduction above. One such methodology fits very closely: Metamorphosis grammars (M–grammars).

M–grammars were chosen instead of other formalisms for the following reasons, most of which have a direct bearing on what we set out to accomplish:

setq	assigns a value to a variable
execute	execute the value of the variable as if it were a statement
readch	input a value into a variable
ty	output a value
cbk	determine if the value of the variable is a valid condition or action (i.e., suitable for inclusion in a codebook as defined for the Barrel/ASP Decision Table Entry, Reformatting,Translation, and Presentation System)
scn	output the contents of an array
push	push a value onto a stack
pop	remove a value from a stack
inqfront	insert a value onto the front of a queue
remqfront	remove a value from the front of a queue
loop/leave/again	looping control structure
if/then/else	traditional if/then/else statement
goto	traditional goto statement
stop	stop execution of the program
size	function; returns the precision if the argument is a number, returns the length if the argument is a character string
concat	function; returns the concatenation of the character string arguments
top	function; returns the value currently on top of the stack
front	function; returns the value currently on the front of the queue
!	plink operator; allows indexing of arrays

Figure 2.2 Summary of some of the Barrel–F Features.

- logic orientation;
- programmable definition;
- ease of understanding;

- complete description of the language;
- easily extended.

M-grammar notation is based on first order predicate logic. M-grammars provide the facilities of other formalisms such as W-grammars and Attribute Grammars but are easier to use and understand. The relational semantics presented here use Prolog as a metalanguage instead of the more traditional lambda calculus. Prolog has a well-defined fixpoint or denotational semantics as well as its proof theoretic semantics, which gives the definitions an adequate mathematical basis [Moss, 1981].

The fact that M-grammars are based on first order predicate logic emphasizes the consistency found throughout this research. Logic is also used in this research project as specifications for a table processing system (see Chapter 3).

Most Prolog systems allow the use of a particular notation of M-grammars (the notation may vary from one Prolog system to another and are sometimes referred to as Definite Clause Grammars [Pereira & Warren, 1980]). M-grammars are parsed by the Prolog interpreter (or compiler) and converted to regular Prolog predicates. Because of this, M-grammars can be executed as if they were Prolog programs. Also, the grammar can be augmented with regular Prolog clauses which can perform extra checking or data manipulation.

Some formalisms provide quite a challenge to the uninitiated in order to understand them (e.g., denotational semantics). M-grammars are somewhat easier to read and understand because first order predicate logic is easy to understand. People familiar with Prolog, a widely known logic programming language, will have little trouble with the definition.

M-grammars can be used to describe both the syntactic and semantic components of a programming language. Thus, a single formalism provides a complete description of the language.

We consider M–grammars to be easily extended. We intend to show the complexities involved in making an extension to the the definition, using as an example the stack and queue statements in Barrel–F which were added after the definition had been completed (see Section 2.2.1).

Appendix 2.2 relates several interesting features of the Barrel–F definition. The most interesting point is the use of continuations to define the goto statement, a technique first introduced by Strachey and Wadsworth [Strachey & Wadsworth, 1974] and commonly used in denotational semantics [Gordon, 1979]. Also mentioned are several shortcomings of the definition. None of these are significant and most pertain to the syntax and not the semantics of the language.

We now give an informal description of some of the details of the M–grammar definition of Barrel–F. The scope of this dissertation does not allow much of a tutorial on how M–grammars and Prolog work. Therefore, some familiarity with them may be necessary to understand the following text. There is excellent documentation available on how the grammar feature of Prolog works [Clocksin & Mellish, 1981; Moss, 1980; Moss, 1982; Pereira, 1983]. Provided below is an example of how the definition can be read and an explanation of the output produced by the execution of the definition. All Prolog and M–grammar examples assume a working knowledge of version 1.4 of C–Prolog [Pereira, 1983].

The definition presented here is modeled after [Moss, 1981]. Appendix 2.3 contains a complete listing of the definition which is conveniently divided into 3 major sections: lexemes (corresponding to the lexical syntax), morpheme (corresponding to the syntax), and sememe (the semantics). The top level relation is shown in Figure 2.3 in Prolog notation rather than M–grammar notation so that we can see the relationship between the 3 parts and their parameters. The top level relation is a predicate with four parameters:

- Text – the text of the program,
- Input – the input file,
- Output – the output file,
- Result – the result of the program.

Normally, Text and Input are provided as input (i.e., they are instantiated when the predicate is invoked) and Output and Result are output (i.e., given values by the program).

```
barrelf(Text, Input, Output, Result) :-
        lexemes(Tokens, Text, []),
        morpheme([Tree | Memory], Tokens, []),
        sememe(Tree, state(Memory, [], Tree, Input, Output, ok),
                     state(Memory1, Continuation, Tree, Input1,
                           Output1, Result)).
```

Figure 2.3. The top level relation of the M–grammar definition of Barrel–F shown here in Prolog notation.

The lexemes predicate takes the text of the program and breaks it up into individual tokens, discarding comments and irrelevant characters. The third argument to lexemes (the empty list) provides the difference list of M–grammars. It effectively says that lexemes should consume all of the text during processing.

The morpheme predicate operates on the tokens produced by lexemes and builds a list composed of two entities: Tree and Memory. Tree is an abstract syntax tree of morphemes (indivisible grammatical elements). Memory is the storage locations named by the program.

The sememe relation involves the abstract syntax tree, the initial state of the abstract machine executing the program, and the final state of the abstract machine. The use of states allows the definition to model the contents of the abstract machine at all times during the execution of the program. The state involves 6 entities:

- Memory – the storage locations produced by morpheme,

- Continuation – the "rest" of the program,
- Tree – the abstract syntax tree produced by morpheme,
- Input – the input file,
- Output – the output file,
- Result – the result of the execution.

All parameters of the state except the Tree change over the course of processing (e.g. Memory vs. Memory1). The contents of the memory will be different as variables are assigned different values. When execution stops the input file will contain only those values not "read" by the program. The output file will reflect any output performed by the program. The Result parameter will reflect any errors encountered during execution. If no errors occur then the Result should be "ok" (the value that we start with in the initial state).

The Continuation parameter is a list of the rest of the statements to be executed by the program. Figure 2.4, which shows the top level sememe relation along with the continuation relation in Prolog format, illustrates the use of continuations. Initially, the continuation list (Cont) is empty (see Figure 2.3 above). We take the first statement to be executed (S1) from the abstract syntax tree and put the rest of the statements (S2) on the front of the continuation list. Then, after the statement is executed, the continuation relation is called which in turn calls the sememe relation with next statement, or list of statements, from the front of the continuation list. The execution of the goto statement simply entails replacement of the continuation list with the statements following the specified label.

We now demonstrate a little bit of the M–grammar notation using the lexemes relation as an example. The top–level relation for lexemes is shown in Figure 2.5.

The argument to lexemes is a list which will be instantiated with the tokens of the program when the definition is executed. The other two arguments that we saw

```
/*** process list of abstract syntax statements ***/
sememe([S1 | S2], state(M, Cont, T, I, O, R), St2) :-
        sememe(S1, state(M, [S2 | Cont], T, I, O, R), St1),
        continuation(St1, St2).
sememe([]) --> [].
/*** continue with next statement on continuation list ***/
continuation(state(M, [S2 | Cont], T, I, O, R), St2) :-
        sememe(S2, state(M, Cont, T, I, O, R), St2).
continuation(state(M, [], T, I, O, R), state(M, [], T1, I, O, R)).
```

Figure 2.4. The top level sememe relation and the continuation relation.

```
/* produce a list of tokens from the list of characters */
lexemes(Tokens) --> [CH], {isnewline(CH)}, lexemes(Tokens).
lexemes(Tokens) --> comment, lexemes(Tokens).
lexemes([Head|Tail]) --> token(Head), lexemes(Tail).
lexemes([]) --> [].
```

Figure 2.5. The top–level relation for the lexical syntax.

in the Prolog notation in Figure 2.3 (the text of the program and the empty list) make up the difference list and are hidden in the M–grammar notation.

One can "read" or interpret the lexemes relation as follows. The first line of Figure 2.5 is a comment. The second line states that if we get a character from the program (the variable CH) and it is a newline character (i.e., the end of a line of text) then we simply call lexemes again with the same argument. Thus, we effectively ignore newline characters in the program by not making them part of the tokens. The third line says that we also ignore comments. The fourth line says that if we find a token in our program then it becomes the head of the list of tokens that we are building. And the rest of the list is built by calling lexemes again. The fifth line says that if there are no more characters in our program then we return the

empty list as our list of tokens. One would have to look further in the definition to see how a comment or a token is defined.

When executed, the definition accepts as input a program written in the Barrel–F language. To facilitate the explanation, we will use the example in Figure 2.6, a Barrel–F program which finds the factorial of a non–negative integer. The comments should make the program self–explanatory.

```
(readch '4' number)|              get a number to compute
(setq i '1)|                      initialize a counter
(setq fact '1)|                   initialize our answer
(if (number > '0) then do)|       work on positive numbers
(loop1)|                          begin our loop
(if (i = number) then leave1)|    have we looped enough?
(incr i)
(setq fact fact*i)
(again1)|                         go to beginning of loop
(endif)|                          end of 1st if statement
(ty 'the factorial of ,number,' is ,fact)
(stop)
```

Figure 2.6. An example of a program to be used as input to the Barrel–F M–grammar definition.

Figure 2.7 shows the output produced when the program in Figure 2.6 is used as input to the definition. The line beginning with "Tokens =" is the list of tokens produced by the lexical syntax portion of the definition. Tokens are derived from groupings of characters. Possible identifiers (i.e., sequential strings of alphanumeric characters) are flagged as arguments to the "id" functor (e.g., id(readch)). Non–alphanumeric characters are left as is. Of course, comments and end–of–line characters are stripped out.

```
Tokens =[ ,(,id(readch), ,',id(4),', ,id(number),), , (,id(setq), ,id(i), ,',id(1),),
          ,(,id(setq), ,id(fact), ,',id(1),), ,(,id(if), ,(,id(number), , >,
          ,',id(0),), ,id(then), ,id(do),), ,(,id(loop1),), ,(,id(if), ,(,id(i), ,=,
          ,id(number),), ,id(then), ,id(leave1),), ,(,id(incr), ,id(i),),
          ,(,id(setq), ,id(fact), ,id(fact),*,id(i),), ,(,id(again1),), ,
          (,id(endif),), ,(,id(ty), ,',id(the), ,id(factorial), ,id(of), ,,,
          id(number),,,', ,id(is), ,,,id(fact),), ,(,id(stop),)]
Tree = [input(id(number)),setq(id(i),val(1)),setq(id(fact),val(1)),
          if(gt(deref(id(number)),val(0)),[loop([],equ(deref(id(i)),
          deref(id(number))),[setq(id(i),plus(deref(id(i)),
          val(1))),setq(id(fact),times(deref(id(fact)),deref(id(i))))])],[]),
          output([val(the factorial of ),deref(id(number)),val( is ),
          deref(id(fact))])]
Mem before sememe = [loc(number,undef),loc(i,undef),loc(fact,undef)]
Enter your input in list form and end it with a period:  [val(4)].
Mem after sememe = [loc(number,val(4)),loc(i,val(4)),loc(fact,val(24))]
Input = []
Output = [val(the factorial of 4 is 24)]
Result = ok
```

Figure 2.7. The output produced by executing the M-grammar definition using the program in Figure 2.6 as input.

The abstract syntax tree, produced by the syntax portion of the definition, is the list following "Tree =" in Figure 2.7. Each line of the program in Figure 2.6 is converted to an abstract syntax statement. Comparison of the source program with the abstract syntax version should make the abstract syntax version readable.

Two of the more interesting abstract statements are the if/then/else and loop/leave/again statements. The abstract syntax for the if/then/else statement is:

```
if (boolean expression,
  [list of statements to execute if true],
  [list of statements to execute if false])
```

In the example, there are no statements to execute when the expression is false (i.e., there is an empty list). That agrees with the logic of the program, i.e., there is no "else" portion of the if/then/else statement.

The abstract syntax for the loop/leave/again statement, where the condition for leaving the loop can be placed anywhere within the loop, is:

loop([statements], boolean expression, [statements])

Again, we note that, in the example, the list of statements before the boolean expression is empty.

The syntax portion of the definition also produces the output beginning with "Mem before sememe =". This is the storage locations of the program, i.e., the list of identifiers and their values (all initially undefined).

Before the semantics portion of the definition is executed the user is asked to enter all input values for the program. This represents the input file, corresponding to the Input argument in Figure 2.3. The values are entered in list format as arguments to the "val" functor. In the example, only one value is entered, 4.

The rest of the output is produced by the semantic portion of the definition. It consists of the values which make up the final state of the abstract machine. The output beginning with "Mem after sememe =" is the storage locations and the values they had when execution of the program ended. The input values which have not been read by the program are displayed after "Input =". The empty list signifies that all values in the input file were read. The output values are shown after "Output =". The line beginning with "Result =" shows the status of the abstract machine when execution stopped. Possible values are: "ok" if execution ended normally, "stopped" if a stop statement was encountered before the end of the program, and "I/O error" if an input/output error occurred.

2.2.1 Making Extensions to the Definition

To demonstrate the extensibility of both the Barrel/ASP system and the M-grammar methodology, both the Barrel-F language and its formal definition were extended by adding functionality in the form of new statements. In keeping with the previously stated "ideal" practice of specification before implementation the definition was extended first.

Rather than add a random set of statements to the language, just for the sake of example, we decided to make the extension meaningful. Statements which manipulate stacks and queues have been touted as useful additions to many programming languages and, yet, are scarcely found. We followed the recommendation of Hull, Takaoka, Jones & Bryant [1985] and Mills [1983] and added four stack manipulation statements, six queue manipulation statements, three stack functions, and four queue functions to Barrel-F.

Modification of the lexical syntax portion of the definition, the lexemes relation, was not required. For every statement and function added to the language, we had to add a relation to the syntax portion, the morpheme relation, of the definition. Additionally, two extra relations were required to register the stack and queue identifiers as variables. The extra relations were very easy to generate and could be added by someone familiar with the definition and M-grammars in less than a days time. For example, the relation for the statement which copies one stack to another, the stcopy statement, is simply:

```
stackstm(Env, stcopy(Stack1, Stack2)) —> [id(stcopy)], [' '],
                                         stidentifier(Env, Stack1, ' '),
                                         [' '], [id(to)], [' '],
                                         stidentifier(Env, Stack2, ')').
```

The relation indicates that the keyword stcopy is followed by a blank, followed by an identifier, followed by " to ", and followed by another identifier.

For the semantics portion of the definition, the sememe relation, we basically needed to add one relation for each of the statements and functions added to the language (several other relations were added but only for readability purposes). Some of these relations were more difficult to develop than the morpheme relations because you have to define the changes made to the state of the abstract machine when the statements or functions are executed. Also, more error checking must occur at the sememe level. Again, we use the sememe relation for the stcopy statement as an example.

```
sememe(stcopy(id(Stack1), id(Stack2))) —>
            lookup(Stack1, val(Max1, Stvals1)),
            ({equal(Max1, undef)},
             stackerror(Stack1, 'not initialized, stcopy ignored');
             {notequal(Max1, undef)},
             lookup(Stack2, val(Max2, Stvals2)),
            ({equal(Max2, undef)},
             stackerror(Stack2, 'not initialized, stcopy ignored');
             {notequal(Max2, undef), length(Stvals1, 0, Len1)},
             (gt(val(Len1), Max2, val(true)),
             stackerror(Stack2, 'overflow occurred, stcopy ignored');
             le(val(Len1), Max2, val(true)),
            update(Stack2, val(Max2, Stvals1))))).
```

First we look up the value of the first stack, make sure it has been initialized, look up the value of the second stack, make sure it has been initialized, make sure the second stack is big enough to hold the first stack, and then update the second stack with the value of the first.

The implementation of the new statements in Barrel-F was very straightforward. There was nothing unusual or problematic about the programming effort.

2.3 IMPLEMENTATION OF BARREL–F

Once Barrel–F had been formally defined we implemented it through the facilities of ASP using the definition as a guide. Before we describe the implementation, we will briefly describe the Barrel/ASP system.

The Augmented Stage2 Processor (ASP) is an extended version of a widely acclaimed general purpose macro processor (Stage2) [Waite, 1973]. The augmentations have allowed us to use ASP as an interpreter. It provides a powerful and flexible tool used for experimenting with and developing programming languages and software systems. ASP is described in detail in Chapters 5, 6 and 7.

Barrel/ASP is a system of applications and pieces of programming languages that have been implemented through the facilities of ASP. The applications focus primarily on table processing. The pieces of programming languages consist of various features or statements from particular programming languages such as C, Lisp, and Basic. They were developed to show the potential of ASP, to provide tools for implementing applications, and to experiment with programming languages. Barrel/ASP is described in detail in Chapters 8 through 12.

The implementation of Barrel–F involved writing macros for each of the Barrel–F statements. These macros are listed in Appendix 2.4. There were also macros written for several other statements which are not part of the Barrel–F language but were used solely to aid in writing the Barrel–F macros. Another set of general purpose programming language statements (called BSYS, see Chapter 9), also implemented through ASP, were used in writing the Barrel–F macros.

The problem of formally proving the correctness of the implementation was not undertaken. It should be possible to use axiomatic semantics to address the problem. Moss discusses using M–grammars to implement axiomatic semantics

[Moss, 1981; Moss, 1982]. It should be relatively easy to implement an axiomatic semantics approach in Barrel/ASP.

2.4 TESTING OF BARREL–F

Testing began before implementation. Fifty–seven programs, listed in Appendix 2.5, were created to test the formal definition. These same programs were used to test the implementation.

While most of the test programs are short and were not designed to be very meaningful, except as a way of exercising all of the features of Barrel–F, an effort was made to include some programs which would be meaningful to a user. For example, one of the test programs computes factorials (see Section 2.2). Also included is a module from the Barrel/ASP Decision Table Entry, Translation, and Presentation System (see Chapter 11). It consists of approximately 80 lines of code. It allows the user to enter the conditions and actions of a decision table and checks for consistency and redundancy.

Many of the programs were designed to test the restrictions of the language. For instance, there are several programs which test the use of the goto statement with the looping control structure. Branches are made into, out of, and within a loop to determine what will happen in those circumstances.

The short length of the programs allows the tester to anticipate the outcome of executing the definition against one of the programs. Thus, the tester is able to visually determine if the test were successful. The same program can be executed against the implementation and the results compared. This helps to assure correctness of the implementation. All of the programs used to test the formal definition were also used to test the implementation.

2.5 FORMAL DEFINITION OF BARREL/ASP

The reasons for providing a formal definition of the tool used to implement Barrel–F have already been given earlier in this chapter. We will now discuss the details of that definition.

The technique of writing macros for ASP is covered in Chapter 5. The following is a brief summary to facilitate the discussion of the formal definition of Barrel/ASP.

ASP macros consist of templates followed by code bodies. A template and code body has to be written for every programming language statement that is to be processed by ASP. A statement submitted to ASP for processing takes the form of a call to one of the macros. The call goes through a pattern–matching process, matching the statement against the set of templates which are currently in ASP's memory. A call is associated with a particular macro if it matches all of the constant part of the template.

Parameters can be passed to macros by specifying place holders within the constant portion of the macro template. The parameters become the portions of the call which do not match the constant portion of the template, i.e., which fit into the parameter place holders.

The number of characters in a parameter can vary. For example, suppose the place holder signifying a parameter is the # symbol and we have a template "hello there, #". The call "hello there, John" would match that template and the parameter would be John. "hello there, Kevin" would also match the template and the parameter would be Kevin.

Each template has a code body associated with it. Code bodies tell ASP what to do when a particular template is matched. Code bodies consist of zero or more lines which are output as text, call other macros, or request that the ASP processor

perform some function. Before any of these activities take place each code body line is scanned for parameter transformations and processor functions.

Parameter transformations, specified by particular sequence of characters placed within the code body line, cause the ASP processor to transform one of the parameters and place the result in the line of text being built. An example is treating the parameter as an arithmetic expression and evaluating it. There are 9 parameter transformations, 6 of which are included in the Barrel/ASP definition (see Figure 2.8).

The other 3 parameter transformations would be easy to define but were omitted because they are not heavily used in practice. For example, parameter transformation number 2 is simply an extension to parameter transformation number 1. The value of the parameter is treated as a symbol to address the memory of ASP. Instead of placing nothing in the constructed line if the symbol is undefined, as is done with parameter transformation number 1, the symbol is given the current value of the symbol generator (a number) and the symbol generator is incremented. It would be a trivial matter to formally define a symbol generator. The parameter transformations and processor functions are described in more detail in Chapter 5.

Processor functions, specified by another particular sequence of characters, cause the processor to perform a function. Processor functions include activities such as looping, input/output, symbol manipulation, etc. There are 18 processor functions, 10 of which are included in the definition (see Figure 2.8). Only one of the undefined processor functions would be difficult to define. The function which adds macro definitions to the ASP memory deals with a portion of the ASP processor which is not described by the formal definition.

When we say we have provided a formal definition of Barrel/ASP what we mean is that we have formally defined the code body lines in Barrel/ASP. Each line

parameter transformations

0 – copy a parameter to the constructed line
1 – copy a string from memory to the constructed line
4 – copy the value of a parameter, treated as an arithmetic expression, to the constructed line
5 – copy a parameter length to the constructed line
6 – reset the value of a parameter
7 – context controlled iteration

processor functions

0 – terminate processing
1 – output a line without rescanning
3 – store information into memory
4 – set skip counter unconditionally
5 – set skip counter conditionally on string equality
6 – set skip counter conditionally on the relative value of two expressions
7 – count-controlled iteration
8 – advance an iteration
9 – escape from processing the current code body
i – input a value

Figure 2.8. The parameter transformations and processor functions included in the formal definition of Barrel/ASP.

in a code body can be thought of as a programming language statement. It will either cause the processor to perform some function, call another macro, or produce a line of text to be output. Thus, in many ways, the definition is similar to definitions of traditional programming languages (like Barrel–F). Where it differs is that all parameter transformations and processor functions in the code body line must be taken care of. Then the definition must decide if the line of text that has been built matches another template or if it should be output.

The only templates known to the definition are those of Barrel–F statements. So, if the line of text that has been built is a Barrel–F statement it is executed. Otherwise, it is output. It is possible to include the Barrel–F statements within the Barrel/ASP definition because most of the Barrel–F statements can be used within the code bodies.

An informal description of the processor functions, parameter transformations, and Barrel–F statements included in the definition is provided in Appendix 2.6. All of Barrel–F except the loop/leave/again, goto, and if/then/else statements is included in the definition. These control structures are not included because the way in which they are implemented precludes their use in code bodies. The lines of the code body are kept in ASP's memory and ASP expects the control structures to be in a file because the control structures actually manipulate the file that the control structure is in. This is not a limitation since they can be functionally replaced by processor functions (and in practice, they have not been missed).

Extending the formal definition to include all of the capabilities of Barrel/ASP would be an intriguing study in the use of formal definitions for something other than traditional programming languages. One would have to model the input of the macro definitions, their subsequent storage in the ASP "memory", and the pattern matching performed when matching a macro call against the templates. We do believe such an extension to be possible, although some major changes to the structure of the current definition would be necessary.

Appendix 2.7 lists the formal definition of Barrel/ASP. It is similar to the formal definition of Barrel–F. Lexemes had to be expanded to recognize parameter transformations and processor functions as tokens. Morpheme had to be modified to handle the lines of text being built, called constructed lines. When a line does not match one of the Barrel–F statements it is treated as a constructed line and passed

on to sememe as such. Sememe evaluates any processor functions and parameter transformations in the line. Then sememe again has to decide, just like morpheme, if the constructed line is a Barrel–F statement. If not it is output. But if it is, then sememe has to give the statement back to lexemes and morpheme to be evaluated as a Barrel–F statement.

The state of the abstract machine changed somewhat from the Barrel–F definition. The Tree entity is no longer needed because in Barrel/ASP there is no goto statement directing you to a specific label so you no longer need to search through the abstract syntax tree for that label. Instead you tell the processor to skip over a specific number of lines (processor functions 4, 5, and 6). An entity to keep up with the skipping had to be added. It determines whether skipping is taking place inside a loop or outside, the depth of nesting of loops, and how many lines are being skipped. Also, an extra file, Chan3, for modeling the output of the constructed lines which do not match another template (in this case, a Barrel–F statement) was added.

Appendix 2.8 details some of the more interesting points about the definition. Mentioned, among other things, are the use of continuations in the definition, the additional output channel added to the definition, and the constructed line concept. Also mentioned are several shortcomings of the definition. None of these are significant and most pertain to the syntax and not the semantics of the language.

As with the Barrel–F definition, a complete suite of 69 programs (actually, in this case, code bodies) which test the Barrel/ASP definition was created. They include programs to test the parameter transformations, processor functions, constructed lines, Barrel–F statements, and Barrel–F statements containing parameter transformations. Some of the programs consist entirely of the high level Barrel–F statements. Some are the same ones used to test the Barrel–F definition.

Some of those were re-written to exclusively use the low level parameter transformations and processor functions. All of the test programs can be found in Appendix 2.9.

Most formal analysis in computer science research has been done on features of high-level languages such as Pascal, Ada, Algol, and PL/1. Little, if any, has been done on features of lower level languages and those of languages such as Forth and the Barrel family where a mixture of low level and high level statements can be exploited as needed. The ability to intermix statements from several levels of the programming language hierarchy is an integral opportunity when writing code bodies within the Barrel family. Code bodies may consist entirely of high level statements such as those employed in the definition of Barrel-F. Or they may consist of low level statements recognized directly by the ASP processor, such as the ASP processor functions. Or, finally, they may contain a mixture of the two. The programmer has flexibility over such aspects as efficiency and readability as opposed to, for example, programming exclusively at a single level in a high-level language.

Furthermore, in also defining Barrel/ASP code bodies, formal methodologies are being applied in an extended mode, especially when the full package of tools, formal methodologies, and applications (such as table processing, neural nets and simulation presented here and in the chapters to come) are considered. We have provided some insight into how to use these formal methodologies for something beyond the usual case of conventional programming constructions. That is, the scope of our formal analysis goes beyond the level of "programming language" by providing a formal basis for an integral part of the overall scheme of ASP processing through definitions obtained from code bodies, along with, of course, the other means of exploiting this style of programming.

CHAPTER 3

RUNNABLE SPECIFICATIONS BASED ON LOGIC

In this chapter we develop the same theme of primacy of formal logic methodology as in the previous chapter. This time, however, we apply it to the potentially much larger domain of applications, i.e., programming systems and environments.

The larger context returns us again to the lifecycle depicted in Figure 2.1. Our point of view in this lifecycle frame is that systems specifications are best given in formal logic. Consistent with this position is that they be developed in a form of logic which has a programming basis. In our case, we use Prolog, a programming language based on first order logic, as the specification language.

Prolog, as a specification language, has several advantages over many other formalisms. Since Prolog is executable on a computer, it allows a more detailed analysis of the specifications at the earliest possible stage in the software lifecycle. The specification can be analyzed and executed before the implementation takes place. That should help pinpoint problems with the specifications without the expense of an incorrect implementation. The specifications can serve as a prototype, or, in many cases, it may serve as an adequate implementation, eliminating the design and implementation phases from the lifecycle.

Prolog also has a firm theoretical foundation with well-defined fixpoint or denotational semantics as well as its proof theoretic semantics [Moss, 1981]. For us, the advantage extends to the fact that we use Prolog in other complimentary studies, i.e., it has in effect been the method of choice for us.

This decision connects our work to prior work, e.g., that of Ruth Davis in her paper on "runnable specifications" [Davis, 1982]. Kowalski also makes a strong case for the use of Prolog as "runnable specifications" [Kowalski, 1984a; Kowalski, 1984b]. We use logic programming to create a runnable specification for an implementation of a decision table presentation processor in Barrel/ASP, considered here as an environment element in a programming system. We note that the developers of the UNIX system seriously considered adding a decision table processor to the overall UNIX system to join other processors such as lex, yacc, m4, graph, and so forth [Johnson, 1982].

The application also connects us with another stream of work [Reilly, Salah & Yang, 1987; Salah, 1986] relating logic programming and Prolog to a range of decision table processing. Weiss and Kulikowski [1984] discuss the sense in which decision table rules may be viewed as an extension to production system rules. If we adopt Minsky's statement that production systems are "just" another way to program a system or the more extravagant claims that, in so–called table–dominant systems, decision tables can be used to replace programming altogether (see [Metzner & Barnes, 1977] for discussion of this and related points), we would be viewing them in programming language terms. Here we view them in the systems or environment context, in part to follow Salah, Reilly and Yang into aiding in the set–up of connections of decision table processors to relational databases and other systems.

3.1 SPECIFICATION

In order for the reader to understand the problem we are trying to solve, i.e., the implementation of a presentation processor, we must informally define a presentation processor. A computerized presentation processor is generally an interpreter which prompts the user to provide input. Upon receiving it, the

processor checks a table, matching the input with table contents, and reports the appropriate output to the user. Typically, the table's conditions (questions) are displayed (presented) to the user; the user selects an appropriate option; and the actions (answers) are presented to the user. We describe presentation processors in more detail in Chapter 10.

car make	cord	cord	reo	reo	duesenberg	duesenberg
condition	good	poor	good	poor	good	poor
commission	5%	1%	10%	5%	variable	variable
shop–work	no–need	3–weeks	no–need	3–weeks	6–weeks	6–weeks
manager–ok	no–req	no–req	no–req	no–req	req	req

Figure 3.1. Representation in table form of a decision procedure relating input values for "car make" and "car condition" to output values entitled "commission", "shop–work", and "manager–ok."

Most of the initial work of developing the specification was done in conjunction with Salah and Reilly [Salah, 1983; Salah, Reilly, & Barrett, 1982]. Informally, the specification states that if we have the decision table shown in Figure 3.1 then we want our presentation processor to prompt us for the "car make" as in:

car make ?

We would respond with an appropriate answer such as "cord." The system would then prompt for the condition as in:

condition ?

We would respond with an appropriate answer such as "good." Finally, the results are returned to us:

commission is 5%

shop work needed is no–need

manager ok is no–req

The prompting should repeat without further intervention by the user. We "escape"

by entering the word, no, in response to the first prompt, i.e., the prompt for car make. Any input of values not in the table produces an error message.

We present in Figure 3.2 a Prolog specification for our presentation processor. The last six clauses are the basic facts for the system, corresponding to the rules of the decision table.

Execution of the specification starts by entering "dt." to call the dt predicate. The Prolog monitor responds with a prompt for the appropriate type of input, i.e., for car make. The user must know the allowable makes of cars. After reading the car make, the system prompts for condition. Again, the user must know the allowable conditions. Obviously, a change could be asked for here, e.g., additional detail in prompting so the user does not have to know all possible values of each predicate place.

After receiving the user's input, the dt predicate calls the intermed predicate, which in turn decides whether to suspend processing or to call the dec predicate. This decision is based on the first of the two intermed predicates (i.e., whether the user enters "no" for car make). If this predicate fails, the second intermed predicate is invoked; it calls upon the dec predicate to process the table.

The dec predicate is the workhorse of the processor. It checks the table (i.e., the table predicates) for a match and outputs the results. If there are no matches it outputs an appropriate message.

An example of the execution, in Figure 3.3, shows that the specification is in–line with our earlier informal description of what we wanted the processor to do. The user starts the action by calling the dt predicate which initiates the prompting for the car make and condition. The correct results are output (as comparison with the table in Figure 3.1 will prove) and prompting is initiated automatically again.

```
dt :- write('car make ? '), read(C1), write('condition ? '), read(C2),
      intermed(C1, C2).
intermed(no, C2) :- write('so long now'), nl.
intermed(C1, C2) :- dec(C1, C2), nl, write('we continue'), nl, dt.
dec(C1, C2) :- table(C1, C2, A1, A2, A3),
               write('commission is '), write(A1), nl,
               write('shop work needed is '), write(A2), nl,
               write('manager ok is '), write(A3), nl.
table(cord, good, A1, A2, A3) :- A1 = '5%',
                                 A2 = 'no-need',
                                 A3 = 'no-req'.
table(cord, poor, A1, A2, A3) :- A1 = '1%',
                                 A2 = '3-weeks',
                                 A3 = 'no-req'.
table(reo, good, A1, A2, A3) :- A1 = '10%',
                                A2 = 'no-need',
                                A3 = 'no-req'.
table(reo, poor, A1, A2, A3) :- A1 = '5%',
                                A2 = '3-weeks',
                                A3 = 'no-req'.
table(duesenberg, good, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.
table(duesenberg, poor, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.
```

Figure 3.2. A Prolog specification of the desired presentation processor.

Two more sets of values are entered before the user enters "no" to end the prompting.

The decision table processor we are implementing in this chapter has a major limitation. The decision table to be processed is hard coded into the processor. To process a different table some simple modifications to the processor must be made. The specification for such a processor would either have to represent decision tables abstractly or it would have to change for each table (the approach we have taken). Abstract representation of a decision table might be adequate for a "pen and paper only" specification but might not be possible in specifications which are designed to be executed on a computer (one of our requirements).

```
> | ?- dt.
> car make ? cord.
> condition ? good.
  commission is 5%
  shop work needed is no-need
  manager ok is no-req
  we continue
> car make ? duesenberg.
> condition ? poor.
  commission is variable
  shop work needed is 6-weeks
  manager ok is req
  we continue
> car make ? no.
> condition ? no.
  so long now
  yes
> | ?-
```

Figure 3.3. A sample dialogue from the execution of the Prolog specification of our presentation processor. The lines beginning with "> " show where the user was required to enter information.

To properly address this problem, we extend the methodology to automatically generate a runnable specification for each decision table to be processed. In Chapter 11 we describe a system which guides the user in developing decision tables. These tables are to be used as input to various decision table processors. We have modified the application so that a runnable specification is automatically generated for each decision table. Chapter 11 covers the details of the system.

A key point of runnable specifications is that the logic program can serve as a presentation processor on a system which has a Prolog processor. In this sense, it could be used directly in the schemes of Salah et al. Our goals are to extend the applicability to a broader context, for example, to contexts in which the C programming language is an integral part of the computational world. The prime applications planned at the time of this writing are in roles relating to simulation environments [Barrett & Reilly, 1987; Reilly & Barrett, 1989; Reilly, Barrett & Lilly, 1987; Reilly, Jones, Barrett, Salah, Strand, Autry & Rowe, 1984] and neural computing [Reilly, McAnulty, Amthor, Wainer, Thurston &Villa, 1987]. Thus, our use of formal specification is to suggest methods for implementing decision tables through the pattern matching facilities of ASP.

Both ASP and Prolog are pattern-directed so it is possible to mimic Prolog constructions in the implementations in the Barrel/ASP system. The closeness of the Prolog specification to possible ASP implementation is supportive of the mimicking activity. As such, the presentation processor is made available to systems for which no logic programming is available, and in a manner that is based on logic expressed in a formal fashion.

3.2 DESIGN

Having decided that the specification was in good order, we moved to the design phase. Ultimately, it should be up to the user to decide whether we are ready

to enter this phase. That is, the user should decide whether the specification meets his or her requirements. If the user is not sophisticated enough to understand formal specifications then the user has a working prototype to experiment with. Execution of the specification is a big bonus at this stage of the lifecycle and may save much effort in the design and implementation phases.

At this point we face the opportunities and constraints that are associated with the solution being with ASP as the implementation tool. One decision is whether to use existing language features developed in Barrel to do the coding. Our goal, that the ASP solution be as compatible with the Prolog specification as possible, is a potential constraint. By not using existing features of Barrel we can write macros to more closely resemble the Prolog specification. The macros, of course, can be written using existing features of Barrel. In fact, we have implemented both solutions and found we much prefer the stronger Prolog resemblance for reasons which will become obvious in the next section.

Recall that the Prolog specification requires that we produce a set of clauses (see [Kowalski, 1979] and [Clocksin & Mellish, 1981] for an introduction to the world of Prolog). The notion of "compatibility" that we accept in the Barrel solution is essentially that we have, in Barrel terminology, a set of "templates" corresponding to the set of predicates in the Prolog specification. ASP utilizes a pattern-directed form of computation that resembles in some ways that in the Prolog processor and yet which is quite different in others (see Chapter 5 for a detailed explanation of how ASP works).

We could demand a higher (second level) compatibility, i.e., that we achieve similar distribution of resources in the sense that the activities of the predicates of the Prolog solution be performed within similarly named definitions in the Barrel/ASP implementation. We try to meet this goal whenever possible but, as we

shall see, it is not always readily achievable. At this point, we might see difficulties in the specification which would cause us to return to the specification phase.

During this phase we have discovered that the Prolog specification has some interesting properties that we had not originally intended to be part of the specification. They are primarily due to the nature of the Prolog processor and, in general, to formal logic but may also be attributed, perhaps, to the way in which the Prolog specification was written. Both instances arise from the ability to execute only a part of the Prolog specification.

The user has the capability to call the dec predicate directly. In that case, no prompting or cycling occurs. The user can also have more than one result at a time displayed by entering a variable name as either the car make or the condition. The user can even display the whole table by entering variables in place of both the car make and the condition.

The sample dialogue in Figure 3.4 illustrates this. The user calls the dec predicate with the second argument (the condition) being a variable. The processor displays the results associated with a car make of cord and a condition of good. The variable name and its value are displayed and the processor waits for input. If the user enters a carriage return, processing stops. If the user enters a semi-colon (as the dialogue indicates), then the processor looks for another match and displays the results associated with a car make of cord and a condition of poor.

The user can also call the table predicates directly, passing any combination of variable or constant arguments. The processor would return those parts of the table that matched the constants or the whole table if all of the arguments were variables.

Because of the relational database style of representing the table within Prolog the user can exercise a Prolog feature called input/output indifference. Normally, the user provides values for the conditions of the decision table (i.e., "car make"

```
>   | ?- dec(cord, Cond).
    commission is 5%
    shop work needed is no-need
    manager ok is no-req
>   Cond = good ;
    commission is 1%
    shop work needed is 3-weeks
    manager ok is no-req
>   Cond = poor ;
    no
>   | ?-
```

Figure 3.4. A sample dialogue from the execution of the Prolog specification of our presentation processor where the dec predicate is called directly. The lines beginning with " > " show where the user was required to enter information.

and "condition"). When the table predicate is called by the dt predicate (indirectly through the intermed and dec predicates) the first two arguments (the conditions) are already instantiated and the last three (the actions) are instantiated as a result of the call. Thus, the first two arguments provide input to the table predicate and the last three provide the output from the predicate. But calling the table predicate directly allows the user to specify the actions of the table and have the processor return the conditions that would bring about those actions. For example, the user could type in:

table(Make, Cond, Comm, "6-weeks", Manok).

to find out what cars in what condition required 6 weeks of shop work. The processor would return all values for Make, Cond, Comm, and Manok that corresponded to 6-weeks of shop work in the table. The user can specify any combination of conditions and actions and those rules that are matched would be returned. Thus, Prolog is "indifferent" as to which arguments are used for input and which are used for output.

The question arises as to whether these "extra" capabilities should be part of the design. If the user who reviewed the specifications is not told about them and is not sophisticated enough to discover them on his own, then it can be argued that they should not be part of the design. However, if the implementor knows about them is he or she required to be true to the specifications and implement them?

This is a question which should be answered during the design phase. We feel that they should be part of the implementation only if it is reasonable to do so. It may be very difficult to implement "Prolog–like" features such as input/output indifference. Indeed, the implementor may be forced to implement a Prolog processor which would not be consistent with our goals, as we shall see below.

3.3 IMPLEMENTATION

Figure 3.5 shows our first attempt at a Barrel/ASP implementation. Appendix 3.1 provides a more detailed listing which includes the supporting macro definitions. In spite of some minor differences which we list below, the implementation is functionally equivalent to the specification. Although we do not use formal techniques to prove this, the language mimicking capabilities of ASP allowed us the make the implementation look almost identical to the specifications. We realize that this may not always be the case, especially for more sophisticated examples, but the ability to execute both the implementation and specification side by side gives us a more rigid method of comparison.

We definitely meet our compatibility objectives since each macro definition is nearly identical to its counterpart in Prolog. We have templates corresponding to each of the Prolog predicates and the processing that occurs within each definition corresponds to that in the Prolog clauses. There are, of course, minor syntactical differences:

- the use of dollar signs instead of commas and periods,

```
dt :-
write('car make ? ') $
read(C1) $
write('condition ? ') $
read(C2) $
intermedprime(C1, C2) $
$
intermed(no, #) :-
write('so long now') $
$
intermed(#, #) :-
dec(#10, #20) $
write('we continue') $
dt $
$
dec(#, #) :-
table(#10, #20, A1, A2, A3) $
write('commission is ') $
write(A1) $
write('shop work is ') $
write(A2) $
write('manager ok is ') $
write(A3) $
$
table(cord, good, #, #, #) :-
#10 = '5%' $
#20 = 'no-need' $
#30 = 'no-req' $
$
```

Figure 3.5. A Barrel/ASP implementation of the desired presentation processor.

```
table(cord, poor, #, #, #) :-
 #10 = '1%' $
 #20 = '3-weeks' $
 #30 = 'no-req' $
$
 table(reo, good, #, #, #) :-
 #10 = '10%' $
 #20 = 'no-need' $
 #30 = 'no-req' $
$
 table(reo, poor, #, #, #) :-
 #10 = '5%' $
 #20 = '3-weeks' $
 #30 = 'no-req' $
$
 table(duesenberg, good, #, #, #) :-
 #10 = 'variable' $
 #20 = '6-weeks' $
 #30 = 'req' $
$
 table(duesenberg, poor, #, #, #) :-
 #10 = 'variable' $
 #20 = '6-weeks' $
 #30 = 'req' $
$
```

Figure 3.5. (continued)

- ASP's use of # symbols instead of variable names,
- the lack of need of the "nl" (new line) predicate,
- the need for a separate line for each macro definition.

One difference lies with the use of the intermedprime definition. This is required because of the difference in the way Prolog and ASP pass parameters.

Prolog passes by value whereas ASP passes the name of the variable. Thus, intermedprime simply calls intermed with the proper values. The definition of intermedprime is very simple and is contained in Appendix 3.1.

Figure 3.6 is an example of the execution of the Barrel/ASP implementation. The same input values were used as with the execution of the Prolog specification (Figure 3.3). It empirically shows that the implementation matches the specification.

If we had decided in the design phase to allow the dec definition to be called directly then one aspect of the specification is not met. Calling dec(cord,Cond) in the Barrel/ASP implementation produces errors (not shown here), whereas the Prolog specification (Figure 3.4) produces all values from the table associated with a car make of cord. This results from two fundamental differences in the Prolog and ASP processors.

The first was referred to above. That is, in Prolog, uninstantiated variables can be passed as arguments in calls to Prolog predicates. If the matching predicate's formal parameter is a constant then the variable is implicitly assigned that value. Barrel/ASP variables must assign values explicitly.

The second difference is the fact that Prolog tries to match all predicates. If more than one predicate matches (as is the case with the dec(cord,Cond) call) then the results from each match are displayed. Barrel/ASP finds the template that best fits the call and does not attempt any more matches.

Since concluding this study, we have considered what might be needed to extend ASP so that it can do what Prolog does in this regard. It is difficult to think of a potentially more interesting and rewarding study. Such a study could be part of a larger study into making ASP a full-fledged logic programming system.

```
    send:
>   dt
    car make ?
>   cord
    condition ?
>   good
    commission is 5%
    shop work needed is no-need
    manager ok is no-req
    we continue
    car make ?
>   duesenberg
    condition ?
>   poor
    commission is variable
    shop work needed is 6-weeks
    manager ok is req
    we continue
    car make ?
>   no
    condition ?
>   no
    so long now
    send:
```

Figure 3.6. A sample dialogue from the execution of the Barrel/ASP implementation of our presentation processor. The lines beginning with "> " show where the user was required to enter information.

In order to simulate the Prolog specification in this respect the dec definition would have to be changed and more definitions (which have no corresponding predicates in the specification) would have to be included. Figure 3.7 shows the change to the dec definition and the four extra definitions. Appendix 3.2 gives the

complete implementation. The dec definition simply calls tableprime which uses pattern matching to determine if dec was called with an uninstantiated variable (which, in the implementation, we specify as anything with an asterisk in front of it). Tableprime makes the appropriate number of calls to the table definition and displays the results (by calling writetable).

The ability of the specification to allow the user to indicate whether to continue to look for more values from the table by entering a semicolon (which is provided by the Prolog processor, not the specifications) is not implemented. It would be a relatively trivial exercise to implement it.

Figure 3.8 shows the execution of the modified version of the Barrel/ASP implementation. Comparison with the execution of the Prolog specification in Figure 3.4 shows how similar the two are.

Another problem arises when you try to call the table predicate directly. Input/output indifference and repeated matching of different table definitions is not achieved. The user can make the call:

table(cord,poor,Comm,Shopwork,Manok)

to find out the commission, shop work needed, and manager's ok relating to a cord in poor condition. But it is not possible to make the call:

table(cord,Cond,Comm,Shopwork,Manok)

to find out all that is known about cords. Nor can you make the call:

table(Car,Cond,10%,Shopwork,Manok)

to find out which cars have a commission of 10%.

These issues have been explored in some detail. Changes to the Barrel/ASP processor could be made to include more "Prolog-like" features, as discussed earlier. Or the table predicates could be modified, much like the dec predicate was, to make them more compliant with the specification. However, we would again

```
dec(#, #) :-
tableprime(#10, #20, A1, A2, A3) $
$
tableprime(#, #, A1, A2, A3) :-
table(#10, #20, A1, A2, A3) $
writetable(A1, A2, A3) $
$
tableprime(*#, #, A1, A2, A3) :-
table(cord, #20, A1, A2, A3) $
writetable(A1, A2, A3) $
table(reo, #20, A1, A2, A3) $
writetable(A1, A2, A3) $
table(duesenberg, #20, A1, A2, A3) $
writetable(A1, A2, A3) $
$
tableprime(#, *#, A1, A2, A3) :-
table(#10, good, A1, A2, A3) $
writetable(A1, A2, A3) $
table(#10, poor, A1, A2, A3) $
writetable(A1, A2, A3) $
$
tableprime(*#, *#, A1, A2, A3) :-
table(cord, good, A1, A2, A3) $
writetable(A1, A2, A3) $
table(cord, poor, A1, A2, A3) $
writetable(A1, A2, A3) $
table(reo, good, A1, A2, A3) $
writetable(A1, A2, A3) $
table(reo, poor, A1, A2, A3) $
writetable(A1, A2, A3) $
table(duesenberg, good, A1, A2, A3) $
writetable(A1, A2, A3) $
table(duesenberg, poor, A1, A2, A3) $
writetable(A1, A2, A3) $
$
```

Figure 3.7. The Barrel/ASP implementation of the dec definitions which allow the user to input variables (beginning with an asterisk) for the conditions of the table.

```
writetable(A1, A2, A3) :-
write('commission is ') $
write(A1) $
write('shop work is ') $
write(A2) $
write('manager ok is ') $
write(A3) $
$
```

Figure 3.7. (continued)

```
        send:
>   dec(cord,*Cond)
        commission is 5%
        shop work needed is no-need
        manager ok is no-req
        commission is 1%
        shop work needed is 3-weeks
        manager ok is no-req
        send:
```

Figure 3.8. A sample dialogue from the execution of the Barrel/ASP implementation of our presentation processor where the dec predicate is called directly. The lines beginning with "> " show where the user was required to enter information.

compromise our previously stated goals and constraints associated with the implementation.

In order to show that the definitions could be expanded to include input/output indifference, Appendix 3.3 provides the Barrel/ASP definitions for a presentation processor which does provide for input/output indifference. This methodology could be employed in the presentation processor implemented in this chapter.

CHAPTER 4

PORTABILITY CUBED: AN EXTENDED NOTION

The microprocessing world is upon us, and with it, the ever present desire to develop on smaller machines, previously successful languages and systems. Languages like C and Prolog are among languages with growing popularity; so are operating systems and programming environments like UNIX.

Since there is such a variety of machines, portability considerations continue to have their importance. With larger systems, particularly those that partition nicely in some logical and functional way, portions of a complete processing activity can be shared among machines. When the various machines can be programmed in a compatible way, system development for network-based solutions is made easier. Maintenance personnel, for example, need not learn several languages to maintain and modify the applications.

We have coined the phrase "portability-cubed" (P3) to describe the unique study of portability offered by ASP. We have combined three instances of portability in one system that can be used in combination or by themselves.

4.1 PORTABILITY OF THE PROCESSOR

First, ASP employs an abstract machine approach to implementation, and thus, is a portable system at the root. The idea of abstract machine modeling is simple and has been discussed thoroughly in computer science literature (see [Brown, 1974, 1977, 1979; Griswold, 1980; Newey, Poole & Waite, 1972; Waite, 1973] for some excellent discussions). It involves designing a machine and a language for that machine that is ideally suited to the application being implemented rather than

trying to fit the application to an existing machine and an existing programming language. This abstract machine would have all the data types required for the application. And the abstract machine language would have all the operations for manipulating the data types that are necessary to implement the application. Writing the application in the abstract machine language should shorten the implementation time.

The ASP processor is written in an extended version of an abstract machine language called FLUB, First Language Under Bootstrap. FLUB was originally designed for the implementation of Stage2. We extended it to aid in the implementation of ASP (see Chapter 6).

Running an application on an abstract machine is an abstract operation. Therefore, a method of running the application on a real machine is necessary. If each operation of the abstract machine language can be defined as a sequence of instructions in an existing programming language on the real machine then the application can be automatically translated to a program that will run on the real machine.

The type of tool normally used to map the abstract machine language to the targeted existing programming language is a general purpose macro processor such as Stage2 or ASP. Macros can be written that will produce appropriate target language code for each operation of the abstract machine language.

Since Stage2 and ASP are both written in an abstract machine language implementing them requires a tool such as Stage2 or ASP. If there exists a running version of Stage2 or ASP (presumably on a different machine than the target machine) then it can be used to translate the FLUB code to the target language for the target machine. This method is known as a half bootstrap.

Perhaps a better method, in which all implementation activity can take place on one machine, is the full bootstrap approach. First, a simple macro processor is implemented and used to translate the FLUB code to the target language. Waite developed a macro processor consisting of about 70 lines of Fortran code that can translate the FLUB version of Stage2 to a target language. ASP requires a more sophisticated macro processor such as Stage2 (or ASP itself) to do the translation; thus, Stage2 must be implemented on the target machine before ASP can be implemented.

Thus, porting the application to a different machine primarily involves re-mapping the abstract machine operations to an existing programming language on the target machine. This is a trivial matter if the same programming language exists on both machines. The application itself does not have to be modified at all.

To port ASP to another machine the simple Fortran macro processor is moved to the target machine, possibly being re-written in another language if Fortran is not available. Then the macros that will translate FLUB to the target language are written (or copied from the other machine if the target languages are the same). The FLUB version of Stage2 is translated to the target language producing an executable version of Stage2 on the target machine. Then the macros that were used to translate Stage2 are re-written for ASP. Two types of modifications to the macros are required. The extensions to FLUB for the ASP processor must be defined. And the extra power of Stage2 over the simple macro processor should be used to provide a more efficient version of ASP.

Porting ASP in this manner is not a difficult procedure. It is certainly less troublesome than converting ASP by hand to another programming language. It is also easier to take care of machine dependencies when they can be confined to a set of macro definitions rather than scattered throughout a large application. The

porting process becomes a matter of understanding the relationship of the abstract machine to the real machine rather than having to understand the algorithm of the application, which may be complex.

ASP has several support routines, written in the target language, which are called by the macro generated code. These are mostly machine and operating system dependent routines such as input, output, and file system manipulation. Porting these routines is by far the most difficult part of the porting process, especially if they must be translated to a different language. But, at the same time, the porting process is made easier by having most of the machine dependencies located in one set of routines.

We have successfully ported ASP to three different machines. It currently runs in the assembly language, MASM, on a Data General Eclipse, in C on a VAX 11/750 running Berkeley UNIX, and in C and a combination of C and Fortran on a 32 processor Sequent machine. Chapter 7 discusses these implementations, as well as the bootstrapping process, in detail.

The portability of ASP gives us the ability to easily move applications written for ASP to other machines without modification. This is an important concept in trying to establish the networking environment that we talk about below.

4.2 PORTABILITY OF LANGUAGES

The second instance of portability within the Barrel/ASP environment involves the language development facility. The user can choose between various programming styles in the implementation of systems. The user can choose a syntax in developing a prototype system that can easily be converted to the form necessary for implementing the real system. For example, the following Lisp statement comes from a Lisp program written for Barrel/ASP.

```
(car (cdr (car (cdr '(w (x y) z)))))
```

The programmer can write the prototype in a style that is close to Lisp. Thus, the prototype can be ported to machines that support Lisp with little or no modification. Features from several different programming languages have been implemented and are available for use in the Barrel/ASP system. These are described in detail in Chapter 9.

4.3 PORTABILITY OF SYSTEMS

Third, in the systems development area, we have the capability of generating decision tables that can be processed on a network of several different machines. We have decision table processors that have been implemented on more than one machine in more than one language. Not only can the tables be ported from one machine to another but several of the processors are portable because of their Barrel/ASP implementation. Tables can be created on one machine and processed on several other machines in several different ways.

When we were exploring the potential of such a network of table processors the machines available to us included a VAX 11/750 running the Berkeley UNIX operating system, a Data General Eclipse, a Prime 400, an IBM 370, and a PDP/11-34. Some of the machines were not physically connected and so the moving of tables and processors from one machine to another was burdensome. The table processors available to us included several complete systems from various sources:

- a processor written in Fortran on the Prime and IBM that translates tables to Fortran code [Reinwald & Dellert, 1968],
- a processor written in COBOL on the IBM that translates tables to COBOL [Dellert, 1972],
- a processor written in C on the VAX that translates tables to C code [Keller & Roesch, 1977; Barrett, 1983a],

- a processor written in Lisp on the VAX that translates tables to Lisp code and vice-versa [Schwartz, 1971];

as well as several locally written prototypical processors:

- a limited entry presentation processor written in assembler and Janus on the Data General and in Barrel/ASP on the Data General and VAX [Barrett & Reilly, 1982],
- an extended entry presentation processor in Barrel/ASP on the Data General and VAX [Minderhout & Reilly, 1982; Reilly, Barrett & Salah, 1982],
- a processor written in Spitbol on the PDP/11 and the IBM that translates tables to Spitbol code [Elrod, 1981],
- several evolving processors written for versions of Prolog available on the VAX [Reilly, Barrett & Salah, 1982; Salah, Reilly & Barrett, 1982],
- an entry, reformatting, translation and presentation system written in Barrel/ASP on the Data General and VAX that translates tables to a form usable by processors in 1, 2, and 5 above and uses graphics in the entry phase [Barrett, 1983a; Barrett, 1983b; Minderhout & Reilly, 1982].

One processor (the sixth on the list) not only translates decision tables to code, but performs the inverse operation of translating code into decision tables. That is, program code written without any thought of tables in mind can be translated into tables. Such inverse translators are considered useful aids to documentation of code, but they can play a larger role than this, when several of them exist within the same system. For example, code developed in Lisp might be translated to tables, which would then be edited for input to a table-to-code system written in Fortran.

This suggests how tables can serve as a focal point for translating algorithms written in one language into algorithms in another. Portability and networking seem well served by such capabilities.

CHAPTER 5

INTRODUCTION TO ASP

5.1 WHAT IT IS

The Augmented Stage2 Processor (ASP) is an extended version of a widely acclaimed general purpose macro processor (Stage2) [Waite, 1973]. Macro processors ordinarily convert strings from one form to another according to a set of definitions. Special purpose macro processors, such as those built into assemblers, transform strings into a particular programming language code. General purpose macro processors like ASP and Stage2 can be used to produce most any kind of text.

Significant changes have been made to Stage2 to produce the current version of ASP. New features have been added and its method of processing has been altered slightly. These relatively small changes have brought about a substantial difference in the characteristics of the processor.

5.2 HOW IT IS USED

Stage2 is described as a processor which accepts character strings as input and transforms them according to a set of definitions provided by the programmer [Waite, 1973]. Typically, the transformation is from one programming language to another.

ASP can still do string transformations, but it performs a slightly different function as well. ASP can interpret the strings as programming language statements and execute them. This provides a powerful and flexible tool for experimenting with and developing programming languages and software systems.

The studies in which ASP has been applied are primarily non-numerical. Stage2 was originally a study in non-numerical processing [Waite, 1973] so ASP is geared toward such problems. The need for such a processor was present in the lab in which the studies took place.

ASP has proven to be a very useful tool for the analysis of programming language features. A control structure, for example, can be implemented and studied in a variety of contexts. Different types of features from different languages can be used in the same environment. Features of a language can be modified in various ways to discover the effect on various environments. New features can be invented, implemented, and tested (see Chapter 8).

The ASP system allows various modes of processing. A particular collection of resources might, for example, operate in a functional style similar to Lisp, for symbolic manipulation and list processing. One study uses ASP to mimic a logic processing style. Features from several different procedural type languages such as Basic, C, and Pascal have been implemented (see Chapter 9).

New languages or variations on existing languages can be developed for use in implementing software systems. One such system implemented in this fashion is a decision table entry, reformatting, translation, and presentation system (see Chapter 11).

Formal studies, with ASP as the central focus, have involved the use of logic programming. Metamorphosis grammars, based on first order predicate logic, are used to formally define a set of language features which are subsequently implemented through ASP. In another study, the logic programming language, Prolog, is used to provide formal specifications for a decision table system which again is implemented through ASP. Decision tables also play an important role in a new concept in portability centering around ASP.

More recently ASP has played a role in the study of simulation environments. The idea is to use ASP for the non-numeric, symbolic component of the simulation environment [Barrett & Reilly, 1987; Reilly & Barrett, 1989; Reilly, Barrett & Lilly, 1987; Reilly, Jones, Barrett et. al., 1984]. In this study, ASP is merged with a combined discrete and continuous simulation language to effect a "combined continuous, discrete, and symbolic simulation" system. The combined system can also be viewed as an expert system with exceedingly powerful numerical facilities. The main use of this facility has been part of a team project on Sixth Generation Computing [Barrett & Reilly, 1988].

5.3 HOW IT WORKS

Two types of input are required by the ASP processor: definitions and calls. Definitions, stored in the processors internal memory, tell the processor what action to take when the calls are received.

The definitions are divided into templates and code bodies. The templates are made up of fixed and variable portions. The variable portions are represented by a special symbol and can be placed anywhere throughout the fixed portions. The variable portions are called parameters and there can be up to nine of them in each template.

The processor does a pattern match between calls that it receives and the set of templates that it has, choosing the closest match based on the fixed portion of the template. The parameters of the template may match any string of characters which are balanced with respect to a pre-defined set of brackets (e.g., parenthesis).

For example, if we had template such as:

GO # LITTLE BOY.

where the # character represents the variable portion (a parameter), and the call to the processor were:

GO HOME LITTLE BOY.

then the match would be made (the fixed portion of the template exactly matches the call) and the first parameter of the definition would be equated to "HOME". The other eight parameters would be undefined. This template would also match a call such as:

GO TO BED LITTLE BOY.

The parameter would be equated to "TO BED".

A code body tells the processor what actions are to be taken when a match is made with its associated template. Within a code body, many activities can take place. Text can be built for output with parameters being inserted into it. Parameters can be manipulated and transformed in several ways. Special functions can be performed by the processor such as looping, assignment of values to variables, input and output, and arithmetic.

It is beyond the scope of this dissertation to give a detailed explanation of the intricacies of the ASP processor. Understanding Stage2 will go a long way towards understanding ASP. A good explanation is provided by William Waite in his writings about Stage2 [Waite, 1967, 1970a, 1970b, 1973] as well as several papers by John Barrett on his early work with Stage2 [Barrett, 1981a, 1981b, 1981c, 1981d; Barrett & Reilly, 1981]. ASP has been described in several papers as well [Barrett, 1982; Barrett & Reilly, 1983]. For the purposes of this dissertation, it will suffice for the reader to have a general knowledge based on the example in Figure 5.1.

The first line of Figure 5.1 defines a set of characters which have special meaning to the processor when used in the definitions and calls. Since the line is included with the definitions the user can specify which symbols the processor should recognize. A period marks the end of templates and calls. The first pound symbol marks the parameters in the templates. Whenever the processor sees a

```
(trim #).
tr(#10)$
$
tr(# ).
tr(#10)$
$
tr(#).
#10#f14$
$
stop.
#f0$
$$
(trim abc  ).
stop.
```

Figure 5.1. An example of ASP definitions and calls.

pound symbol in a template it looks for a matching balanced string in the call. The dollar sign is used to delimit the end of the code body lines. A dollar sign on a line by itself marks the end of a definition (with a new template to follow). Two dollar signs on a line by themselves mark the end of all definitions and signals the start of calls. The second pound symbol is used within the code bodies to indicate special tasks for the processor to perform (known as parameter transformations and processor functions). The last seven symbols – space, right parenthesis, plus, minus, asterisk, slash, and left parenthesis – tell the processor what symbols are being used for space, parenthesis, and the four arithmetic symbols.

The second line of Figure 5.1 is the first template of the definitions. Based on the information in the previous paragraph the reader can see that there are four definitions and two calls in the example. The four templates are (trim #), tr(#), tr(#), and stop. The two calls are (trim abc) and stop. Code bodies can consist of any number of lines although for this example each of the four code bodies has only one line.

Figure 5.1 demonstrates two of the most powerful features of ASP: recursion and pattern matching. To demonstrate this, an explanation of the processing that takes place is in order.

The first call matches the first template – (trim #). The parameter is equated to "abc " (including the two spaces). The code body is processed by scanning each line from left to right, looking for the special character (#) which signifies a parameter transformation or processor function. A line of text is built from the non–special characters. When the processor gets to the pound symbol it looks at the next character. If it is a digit then a parameter transformation is being requested. The digit specifies which of the nine parameters to transform. The second digit after the pound symbol specifies which transformation to perform. In most cases, the value of the transformed parameter is placed in the line of text that is being built.

The #10 in the example tells the processor to perform transformation zero on parameter number one. Transformation zero specifies that nothing is to be done to the parameter so it is placed in the line of text as is. Thus the line of text that is built is "tr(abc)". This line is automatically submitted as a call to the processor and a match is made with the second template – tr(#). It does not match the third template because more characters can be exactly matched with the second template (because of the space character). This time the parameter is equated to "abc ". The line of text that is built from the code body is "tr(abc)". This is submitted as a call to the processor and matches the second template (recursion). The parameter is equated to "abc" and the line of text that is built is "tr(abc)". When this is submitted as a call it matches the third template (since there are no spaces in it). The #10 in the code body places the parameter – abc – in the line of text. The next pound symbol is not followed by a digit so it is treated as a processor function. The digit following

the f means to perform processor function number one which causes ASP to write the text that has been built to the input/output channel specified by the character after the 1. So in the example, the parameter is written to channel four which is the users terminal. Thus, what is written on the users terminal is "abc".

Since there are no more lines in the code body the recursion unwinds and processing for the "trim" call is finished. The next call to be processed is "stop". The code body for that template says to perform processor function number zero which halts the processor.

ASP has several different channels through which it can perform input and output. By definition, when ASP is executed, it looks for definitions on channel number one. When all the definitions have been read ASP expects to receive calls. One of the processor functions tells ASP to get the next call from a different channel. The channels can be tied to various input/output devices. Thus calls can be issued from the users terminal (an interactive environment) as well as disk files (a batch environment).

Calls, as we have seen, can also be issued from within code bodies. Hierarchies of definitions can be built which allow calls to model very high level programming language statements. These high level calls can be used within the code bodies, making the task of programming the code bodies much easier. Programmers can also choose a relatively low level of coding style, such as that of using parameter transformations and processor functions directly in the code bodies. Normally, the programmer will use a mixture of high level calls and low level constructs in programming the code bodies.

Both top–down and bottom–up modular approaches to the development of definitions are facilitated by the very nature of ASP. In our research we first

developed a basic set of general purpose definitions. These definitions (described in Chapter 9) are used in almost all of our applications.

CHAPTER 6

DESIGN CONSIDERATIONS OF ASP

6.1 CHOICE OF STAGE2 AS A BASE

Several factors led to accepting Stage2 as the focal point of this research. Many of them had to do with the environment in which the work was taking place as much as with Stage2 itself. These included:

- a concern for small machines;
- a lack of non-numerical software;
- a need for portability among several machines;
- a desire to study programming language issues.

At the time of inception of this project we had access to a Data General Eclipse S/130 as well as limited access to other similar machines, e.g., a PDP-11/34 and a Prime 400. A larger-scale IBM 370 was also available. Although we wished to ultimately work with all of these machines in a networking environment we knew that most of the work, at least initially, would be done on the Data General.

The Data General was surprisingly deficient in software, especially non-numerical software in which we were most interested. It was useless, for example, in courses which utilized Snobol and Lisp. In addition, we were interested in prominent new languages and systems, such as Prolog and Icon, and perhaps less well known languages such as Pop-2 and Logo.

We implemented Stage2 on the Data General as a first step in remedying the non-numerical software deficiency. Several attributes of Stage2 were instrumental in our decision to use it:

- orientation toward non-numerical software,
- ease of implementation,
- history of use,
- portability.

An initial Stage2 processor is easily implemented on a variety of machines. We have implemented it on two different machines in five different languages [Gibson, 1982]. It also has a lengthy history of usage elsewhere, both in the U.S. and abroad. Twenty-five implementations are mentioned in a 1972 paper by M. C. Newey [Newey, Poole & Waite, 1972]. It has continued to be used from time to time for various purposes (e.g., [Papakonstantinou, 1980]). Even today a slight modification of Stage2 called TILT (Texas Instruments Language Translator) is being used internally by Texas Instruments as a software development tool [Wixson, 1986].

A number of non-numerical software systems have already been developed with Stage2 including a version of Lisp, a string processing language called Wisp [Waite, 1973], and the "universal assembler" Janus [Coleman, Poole & Waite, 1974; Haddon & Waite, 1978a]. Stage2 has a firm basis in operational semantics through its abstract machine implementation which also provides a strong portability emphasis. Stage2 is a curious computer science item in its own right. It is a very powerful and capable tool.

ASP was derived from Stage2 because of a portability problem. There was a decision table manipulation program written in Snobol running on the PDP-11. We wished to port the program to the Data General because of difficulty in accessing the PDP-11. Stage2 seemed a likely candidate for translating the Snobol code to a language available on the Data General. However, it soon became clear that Stage2 almost had the capability to act as a Snobol interpreter on a very limited scale. We quickly had several Snobol-like statements being executed by Stage2.

However, the methodology proved to be awkward in several instances and it again became clear that if we were going to continue to use Stage2 as an interactive interpreter that some features of Stage2 would have to be changed and other features would have to be added. Thus, ASP evolved.

6.2 EXTENSIONS AND MODIFICATIONS

The extensions and modifications that have been made to Stage2, to arrive at ASP, include eight new processor functions and four changes in processing style. The new processor functions are described below. Appendix 6.1 gives a more technical description of the new processor functions in the same format that Waite uses to describe the original processor functions [Waite, 1973] (also, see [Barrett, 1982] for a discussion).

1. add-definitions function

Stage2 has a static definition loading scheme. Stage2 expects definitions and then calls. Once calls are begun no new definitions are allowed. ASP has a dynamic definition loading scheme. The add-definitions function allows the user to intermix the presentation of definitions and patterns to the ASP processor. Thus, ASP allows definitions, then calls, then any order of definitions and calls.

There are three factors which make this an important addition:

- interactive modification of definitions,
- interactive modification of the environment,
- efficiency.

In Stage2, if a definition contains an error the user must exit Stage2, modify the definition, and start over again. In ASP, the definition can be modified on-line, without leaving ASP. An editing facility has been designed and prototyped that will allow a user to edit an existing definition or create a new definition and add it to the set of definitions that are known to the ASP processor. Thus, the user can correct

mistakes and/or create and test new definitions without ever leaving the current ASP session.

Most creation of ASP definitions has taken place in a modularized fashion. That is, non-similar definitions are placed in different files. For example, there is a file of definitions which mimic Lisp functions and a file of definitions which mimic Logo functions (see Chapter 10). Upon start up of an ASP session, the user can choose which definitions should be loaded into the processor. If, in the middle of the session, the user discovers that he or she would like to use a definition that has not been loaded then the file that contains that definition can be loaded and the session continued. Otherwise, the user would have to exit the session and start a new session, loading the proper files of definitions.

Many times the calls to the ASP processor are in the form of a program that runs as a batch job and the definitions that are needed are dependent on the input data. Rather than load all the definitions, the program can include code which will examine the input and load the proper set of definitions. This can improve the execution speed of the program since there would be fewer definitions to match with the calls. On machines which have a limited address space such as the Data General there may not be room in memory for all of the definitions required by a large ASP program. In that case the programmer can programmatically configure an ASP session so that only those definitions that are going to be used have to be loaded.

2. close function

It allows the user to disassociate a file from an I/O channel. An I/O channel is defined as an association between a "logical device" available to the program and a physical I/O device [Barrett & Reilly, 1981; Orgass & Waite, 1969; Waite, 1970a, 1973]. The version of Stage2 that we began this project with had four channels over

which I/O could be performed with a fifth channel recommended and easily obtainable [Waite, 1973]. These channels have to be used for specific purposes most of the time such as input of definitions and input of calls. They are tied to a particular device or disk file at the beginning of a Stage2 session and remain so until the session ends. The need for a close function did not exist with only four dedicated channels.

We felt that more channels were needed with ASP as well as a more flexible I/O scheme so an additional 30 channels were added. These channels are more flexible because they are not associated with a particular device or disk file until a call is made to the proper processor function. Without the close function, such associations, once made, would have to remain in effect throughout the session. The close function becomes especially useful with operating systems such as Berkeley UNIX 4.1 which allows a maximum of 20 open files per process.

3. graphics function

It provides an interface to a graphics terminal. Graphics is an important component of many of the types of studies in which ASP has been involved, especially table processing and simulation environments (see Chapter 12). The interface allows the user to enter DEC Regis commands on a GIGI terminal (other types of graphics terminals could be easily incorporated into ASP). The user has the option of saving the graphics commands in a disk file so that pictures that are built interactively can be saved. Sophisticated graphics macros such as box, grid, and circle can be built through ASP's hierarchical definition facility.

4. input function

It allows input of values for ASP variables. Stage2 only provides for input of definitions and calls. In studying programming languages, a method was needed to provide input of data so that traditional languages could be modeled. Before the

input function was developed, data input was achieved by fooling ASP into thinking it was inputting a macro call. Then the call was intercepted and assigned as a variable value. This method worked well unless the call happened to match a definition template in which case the definition was executed instead of the data being assigned to the variable.

5. escape function

It provides a method of temporarily escaping from ASP to communicate directly with the operating system. A subprocess is generated which executes the command language interface of the operating system. We view ASP as a tool to be used in conjunction with other software products. This function provides a method of integrating those products with ASP.

6. system function

Like the escape function above, it provides a method of executing an operating system command from within the ASP process. However, only one command is executed and then control is immediately returned to ASP. This function is useful in non–interactive ASP programs where an automatic return to ASP is needed.

7. bedit function

It provides a method of using a text editor without terminating the ASP session. It causes a separate ASP session to be started with the definitions of a text editor called bedit loaded (see Chapter 10). Thus, one can do some editing in the middle of an ASP session and pick back up with the ASP session right where he or she left off.

8. trace function

It allows the tracing of all calls made to the ASP processor. We have incorporated a trace facility into ASP which the trace function turns on and off.

When turned on, all ASP calls are displayed on the users' output device. This function is critical to being able to easily debug complicated definitions.

The four changes in processing style involve several different types of modifications to the Stage2 processor. First, additions to the abstract machine language in which ASP is written were made. Eight new statements were added to the language. Second, the abstract machine language program which implements ASP was modified. Some of the abstract machine language statements are implemented by calling subroutines which carry out the function of the statement. These subroutines are written in a language known to the host machine (e.g., assembly language). The third type of modification is in the form of changes to the subroutines necessary for implementing Stage2 and the addition of new subroutines required for the implementation ASP. The four changes in processing style are described below.

1. ability to reference over 30 files simultaneously

Stage2 is very limited in its ability to simultaneously reference several different files for input or output of data, macro calls, or definitions (see the description of the "close" processor function above). ASP needs a more flexible I/O scheme than Stage2; for example, to allow input of definitions from several different files (see the description of the "add definitions" processor function above). So 30 additional I/O channels were added with the ability to associate each channel with a file.

2. ability to replace definitions with new ones

In Stage2, if several macro definitions have identical templates then the definition that is used when a call matches that template is the definition that was input first. In ASP the opposite is true – the definition that is input last is used. This strategy is essential to interactively modifying or replacing definitions (see the definition of the "add definitions" processor function above).

3. ability to use file names rather than channel numbers

The processor functions which perform input/output operations require the designation of the number of the channel over which the operation is to be done. ASP allows a file name to be used instead of the channel number. A channel is assigned to that file automatically if one is available. Using a name instead of a number makes it much easier to keep up with what the channel is being used for.

When two or more channels are associated with the same file, use of the file name will result in the lowest numbered channel being chosen for the input/output operation. For example, if a file is associated with channel 6 and then the same file is associated with channel 7, any input/output operations requested by file name will result in the input/output being performed on channel 6. The channel number of the file is made available to the user so that when an input/output operation needs to performed over channel 7 (presumably the file pointers will be pointing to different records in the file) the channel number can be used rather than the file name.

4. ability to use parameter transformations in previously forbidden contexts

With some processor functions ASP will evaluate parameter transformations to get its arguments where Stage2 will only allow constants. This is critical to the situation described in the previous paragraph. When a file name is used with input/output processor functions the number of the channel which ASP assigns to the file is placed in one of the nine parameters. In order to use that channel number with subsequent input/output processor functions a parameter transformation has to be used.

Most of the extensions and modifications play a major role in changing the processor from batch–oriented to interactive and batch mode. The interface to graphics processing allows ASP to be used for various new applications, which are

expedited by graphics, e.g., table processing. Operating system facilities that can be used include creating multiple processes, e.g., allowing shared processing of applications and use of system tools such as editors and command line interpreters (CLIs) — as was done with the DG's CLI.

In summary, these extensions and modifications have produced:

- a new, highly interactive framework for ASP users,
- the ability to dynamically change the environment of an ASP session,
- on-line creation of new patterns and definitions,
- on-line modification of existing definitions,
- code bodies with more transformation abilities,
- operating system facilities made available,
- graphics usage,
- extended file handling,
- debugging capabilities,
- convenience of using symbols instead of numbers.

Overall, this modernizing of an "old" product and allowing use of it in new ways is a goal often advocated in computer science but rarely achieved.

6.3 ANALYSIS OF ASP USING DATA FLOW DIAGRAMS

To analyze and highlight the differences between Stage2 and ASP, we have used data flow diagrams [Gane & Sarson, 1979]. Two such data flow diagrams, one for Stage2 in Figure 6.1, based on the version found in [Waite, 1973], and one of ASP in Figure 6.2, provide a demonstration of some aspects of the changed world of ASP relative to that of Stage2.

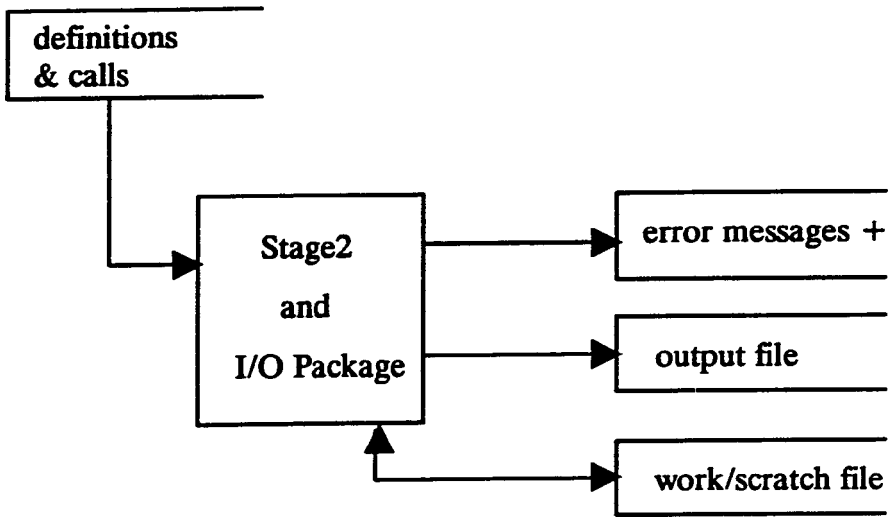


Figure 6.1. Data flow diagram of Stage2.

Three differences stand out in a comparison of the the two diagrams. The first is the addition of the user to ASP processing. This reflects the orientation of ASP to interactive rather than batch processing.

The second is the addition of 30 files providing more flexibility in I/O operations. These files provide the necessary means for the interpretation and compilation of the kinds of languages we have been studying through ASP (see Chapter 8). These are also necessary for table-processing applications as well as numerous other possible applications.

The third difference is the separation of the files providing the definitions and the calls. Again more flexibility is obtained in ASP especially with the capability of mixing the input of definitions with calls (see Section 6.2).

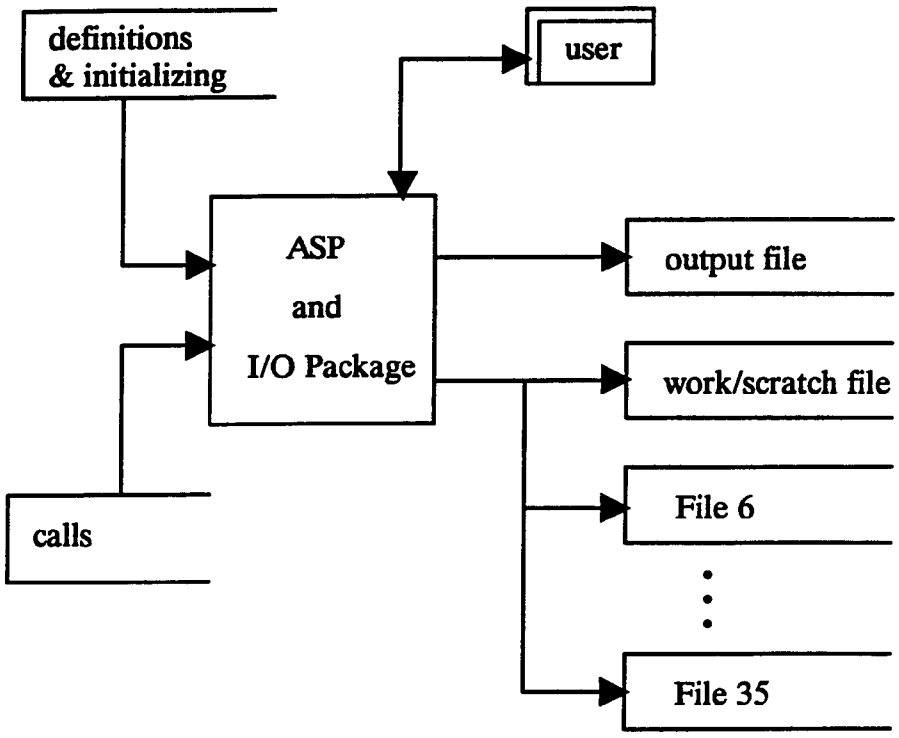


Figure 6.2. Data flow diagram of ASP.

CHAPTER 7

IMPLEMENTATIONS OF ASP

Stage2 can be implemented by a full bootstrap; i.e., a simple processor is implemented by "hand" and then used to obtain a more complex processor [Barrett & Reilly, 1981; Waite, 1973]. With this method of implementation the complex processor can exist in the form of a program for an abstract machine written in an abstract language designed especially for that machine. Thus, the machine and the corresponding machine language can be designed in a manner which will ease the implementation of the processor. The implementation can avoid dependencies on any particular real machine and need not be tied to the availability of a particular programming language compiler. Only one version of the processor written in the abstract machine language need exist; there need not be a version for every machine and every programming language in which it is to be implemented.

There does exist, however, machine dependent portions of the processor in the form of subroutines written in a real programming language (e.g,. FORTRAN). Therefore, the only effort in porting the processor to a machine which does not support a compiler for programming languages in which the processor has already been implemented in is re-writing the machine dependent subroutines and the simple processor. The complex processor does not have to be modified before moving it to a new machine.

The simple processor used to implement Stage2 is called SIMCMP. It is basically a program for pattern matching and replacement. Pattern matching is used to recognize the statements in the abstract machine language called FLUB

(First Language Under Bootstrap) and replacement is used to construct the programming language code for the target computer. Both processes are stripped to the bare essentials since SIMCMP is to be implemented by hand [Waite, 1973].

The implementation of ASP carries this two stage bootstrapping process one step further. The SIMCMP program is not sophisticated enough to handle the needs of ASP translation. Stage2 is the processor used to translate ASP to the target machine programming language. ASP is written in an extended version of FLUB (see Appendix 7.1 for details). Appendices 7.2 through 7.4 contain all of the programs necessary for a C language implementation of ASP including a listing of the FLUB version of ASP, the macros needed to translate it to C, and the C version of the support routines.

This puts a further strain on the portability of ASP because Stage2 must be implemented on the target machine first. But the constraint is not considered serious since Stage2 is relatively easy to implement. In our lab Stage2 has been implemented on two different machines in five different programming languages.

Several files of macro definitions which test the implementation are provided with Stage2. These files can be used to test the implementation of ASP as well. In addition, I have written 42 sets of macro definitions to test the features of ASP which are different from Stage2.

7.1 DATA GENERAL ECLIPSE

The target programming language for ASP implementation on the Data General Eclipse S/130 is the macro assembly language (MASM). The processor consists of approximately 7800 lines of MASM code. A small portion of the code is in the form of hand coded subroutines (which implement the machine dependent portions of the processor) but the majority is generated by macro translation of the abstract machine language version of ASP.

7.2 VAX 11/750

The current version of ASP on the VAX consists of approximately 2500 lines of C code. As with the Data General most of this C code is not hand-written; but is generated by macro translation from the extended version of the abstract machine language called FLUB. The (extended) FLUB code for the current version of ASP is approximately 1200 lines long and approximately 50 macro definitions are used to carry out the translation from FLUB to C (see Appendix 7.2 for the FLUB code, Appendix 7.3 for the macros, and Appendix 7.4 for the C hand coded subroutines).

7.3 SEQUENT 21000

There are two versions of ASP available on the Sequent. The first is a verbatim copy of the C version which runs on the VAX. No changes were necessary to either the macro definitions or the support subroutines.

The other version is a combination of FORTRAN and C. The macro definitions were re-written to produce FORTRAN code instead of C. The resulting FORTRAN program has about the same number of lines as the C version. The support subroutines remained in C so we could avoid the effort of re-coding them in FORTRAN. Some minor modifications of the subroutines were made to allow for the conventions required for passing arguments from FORTRAN to C on the Sequent. By using FORTRAN we wished to position ASP for future enhancements which could take advantage of the multiprocessing capabilities of FORTRAN on the multiprocessor Sequent.

CHAPTER 8

LANGUAGE FEATURE ANALYSIS BY ASP IMPLEMENTATIONS

A number of experiments have been performed with ASP on isolated or small groups of features of programming languages [Barrett & Reilly, 1984]. For example, experiments have been done to:

- analyze various aspects of certain language features;
- analyze the effect of the programming environment on the use of a particular feature;
- determine the capabilities of implementation within ASP;
- determine the needs of the underlying processor in carrying out such implementations.

Some of the programming language constructs involved in these experiments include:

- the INPUT statement from Basic;
- a couple of pattern matching statements from Snobol;
- various forms of control structures, e.g., WHILE, REPEAT and CASE modeled after appropriate constructs in languages such as C, Pascal and Ada;
- the "plink" ("!") operator notation for arrays from BCPL;
- and the projection operation from the topic of relational databases.

For example, while implementing the array operator from BCPL we noticed that ASP would easily accommodate character string subscripts and sparse arrays

such as those found in MUMPS [Walters, Bowie & Wilcox, 1982]. This new combination of language features forced us to develop a new syntax for the array operator. A second example is the implementation of the INPUT statement from Basic which led to the development of the input function of ASP (see the description of the input function in Chapter 6).

An important aspect of language feature analysis is the ability to implement language features which are proposed in various scientific publications. In many cases implementation of these proposed features is not done. Elliot Soloway has proposed the use of a new looping control structure which he claims has a closer "cognitive fit" with an individual's preferred cognitive strategy in programming [Soloway, Bonar & Erlich, 1983]. The looping construct allows an exit from the middle of the loop as opposed to the more traditional exit from the beginning or end of the loop.

In this case the implementation was carried out by Soloway; the control structure was included in a version of a Pascal compiler. The amount of work required to do this implementation was not mentioned but modifications to a compiler are not normally trivial. The implementation of this construct in ASP took a matter of minutes. We were able to study its use in a much broader context than just Pascal and we found it to be useful. It has been included in our "core" set of language features as the major looping construct and a formal definition of it is provided (see Chapter 2).

Experimenting with Soloway's control structure in a real environment forces one to deal with unforeseen issues. For example, how does such a structure interact with branches; that is, does it make sense to branch into it and out of it? The answers to such questions help bring into focus the reality of whether such structures are in fact useful or practical.

In fact, many experiments can be performed on such language features in the ASP environment. The syntax of statements can be changed with a few key strokes in a text editor (by changing the template of its definition). Statements can be placed in environments for which they were not intended; e.g., procedural statements in a functional programming environment. Properties can be given to or taken away from various statements. And all such experimentation can be empirically carried out in an environment set up specifically for it.

In implementing experiments like these, and, in fact, for all implementations in ASP, the user has a wide choice over the level and nature of the instructions to be used, both in the code bodies and in the environment in which the experiment is to take place. Low, intermediate, high and very high level instructions can be employed. At the lowest level, code bodies in ASP are programmed at the abstract machine language level (see Chapter 6). The fact that code bodies can include calls to other definitions means that hierarchies of instructions can be built by building successively higher–level layers. Since, in ASP code bodies, we can intermix levels of coding, the process of developing (new) instructions is quite flexible, i.e., we can make adjustments for readability and efficiency, as the situation warrants. Likewise, since ASP allows access to support routines written in any language easily accessible by ASP, e.g., C on the VAX, still other possibilities of adapting the level and nature of instructions to the situation exist.

CHAPTER 9

KITS

Having developed such a system as ASP, a first use of it is to demonstrate potential. We collect such demonstrations into a loose coalition of definitions for ASP denoted as Barrel. We refer to these definitions as "kits" since they implement collections of related programming language features which can be used by themselves or assembled together to form interesting, dynamic and powerful programming environments and testbeds for experimentation. Most kits are modeled after existing languages or mixtures of existing languages, containing, for example, all the control structures of a language, or a mixture of popular control structures from several languages, with a purpose of studying them as a unit.

Barrel, like ASP, has certain constraints placed on it. Foremost are those concepts and recommendations emerging from computer science. We interpret this to include such items and ingredients as: a formally described subset of Barrel; relationships with concepts such as logic programming; endorsement of capabilities which help span the system lifecycle. All of these concepts are discussed thoroughly in Chapter 2.

The kits can be divided conveniently into interpreted and compiled ones, and they are described below.

9.1 INTERPRETED KITS: BARREL

Barrel consists of several kits programmed through the use of the facilities of ASP [Barrett & Reilly, 1984; Minderhout, Reilly, Barrett & Gibson, 1982]. A distinguishing feature of most of these kits is that ASP facilities are used

interpretatively rather than providing compilation capabilities. The kits we now describe are:

a) BSYS, features used in implementing most of the other kits,

b) BBAS, a Basic–like interactive kit,

c) BLISP, a subset of the list processor, Lisp,

d) BICON, the hub of the Icon string processor,

e) BGIGI, an interface to GIGI/Regis commands,

f) BCNTRL, while and repeat statements,

g) BCASE, a case statement.

A listing of the statements included in each of these kits can be found in Appendices 9.1 through 9.7.

BSYS includes features which are among those most frequently used in the development of code for ASP. As other kits were developed there arose a set of features which were being used over and over in almost all of the kits. These features were placed in BSYS and most of the other kits are implemented by using features from BSYS. Thus, features in BSYS share characteristics such as general functionality, efficiency, and ease of use. In many cases the BSYS commands provide a convenient alternative to directly using the machine language level facilities of ASP (i.e., parameter transformations and processor functions; see Chapter 5). The types of features found in BSYS include assignment statements, flow control statements, input and output statements, and stack control statements.

BBAS, like BSYS, includes many features that are basic to the operation of Barrel, that is, much of the software developed through ASP depends on BBAS for its operation. BBAS is functionally at a higher level than BSYS and has a wider variety of statements. BBAS resembles any of a number of interactive systems such as, e.g., Basic. However, it has structured control statements, some modeled after

Ada, Pascal, and C, and is therefore more like a structured Basic such as COMAL [Gratte, 1984]. Some of the types of commands found in BBAS include input and output statements, file processing commands, flow control commands, array processing commands, string processing commands, assignment statements, macro defining commands, commands which interface to the operating system, and commands which trace the flow of execution of Barrel code.

BLISP consists of a subset of Lisp, including the basic primitives of mathematical Lisp (CAR, CDR, CONS, EQ, and ATOM). BLISP is capable of full nested combination evaluation, e.g., expressions of the form (cons (car l) (cdr m)). The implementation of combination evaluation is similar to Burge's method [Burge, 1975; Eades, Reilly, Barrett & Minderhout, 1982] of decoding or compiling and then interpreting or executing, using two stacks to hold intermediate results.

Other built-in functions and support routines in BLISP are implemented as part of the environment in which the processor resides. For example, function definitions rely on ASP's definitional facility as opposed to a separate define statement.

BICON provides string processing facilities modeled after those of the programming language ICON [Griswold, 1982; Griswold & Griswold, 1983]. ICON is a product resembling Snobol but has a different set of primitive constructions and a new approach to pattern matching. It also embeds the string facilities within a Pascal-like higher-level language framework. Included are commands such as upto, many, any, and move.

BGIGI includes commands which allow the use of a GIGI/Regis graphics system. We primarily use graphics to generate and display tables from within the table processing component of Barrel (see Chapter 12). We have implemented some of the turtle graphics primitives of Logo as well (see Chapter 10). And we

have also devised a package with which the user can interactively build a picture on the terminal while saving the commands on disk.

BCNTRL contains two different kinds of control structures: a "while do" statement and a "repeat until" statement. BCASE contains an implementation of a case statement.

A simple example of a program which can be interpreted by these kits might give the reader an idea of the kinds of things that can be done with them. The program in Figure 9.1 provides the user with a means for interactively executing single statements from any of the kits that have been loaded into the ASP system for the current session. A line is read from channel 4 (the user's terminal) and then executed. The program requires the definitions from the BSYS and BBAS kits.

```
(fty 'we permit 100 executions only)
(fty ' all submitted code should start in column 1)
(for i := 1 to 100 begin)
(fty 'send:)
(readch '4' code)
(fexecute code)
(end for)
stop
```

Figure 9.1. An example of a program which can be interpreted by the BBAS kit.

9.2 KITS AND COMPILERS

Two kits have firm connections with compilers. One of these is for Janus, and the other for a compiler version of a purely function subset of Lisp with which ASP communicates.

9.2.1 Janus

The Janus programming language is described by W. Waite and his colleagues as a universal intermediate language [Coleman, Poole & Waite, 1974; Haddon & Waite, 1978a; Waite, 1976, 1978]. The level of Janus as a language lies between high-level and assembly language. It was designed to serve as the target language for the translation of high-level languages. It may be viewed as the assembler for the Janus abstract machine. Because of the abstract machine approach, Janus has the potential to be a highly portable language. Thus, programs written in a high level language on one computer can be translated to Janus and subsequently executed on a number of different computers.

We have used ASP to translate Janus to the assembly language of a Data General computer [Barrett & Reilly, 1982]. Approximately 170 definitions comprise the translator for Janus. The definitions are submitted to ASP along with the Janus program to be translated to produce an assembly language program. This process is illustrated by the data flow diagram in Figure 9.2.

We have dealt mostly with the basic aspects of the implementation of Janus, especially as they relate to the Data General, e.g., such topics as procedures, input-output, character strings, and numbers. We have concentrated on pragmatic issues, e.g., practicalities of portability, direct versus indirect translation of Janus, and potential usages of the system. We were also able to convert a decision table processor, previously written in the assembly language of the Data General to Janus. Rewriting it in Janus was done, in part to make it portable and, in part, to gain insight into comparative programming in Janus and in a "real" assembler.

Theoretical issues, which may in the future be relevant to us, are discussed by Haddon and Waite [1978b]. These include Janus' tree-based memory system and abstract machine approach, similar to ones exploited in operational semantics of

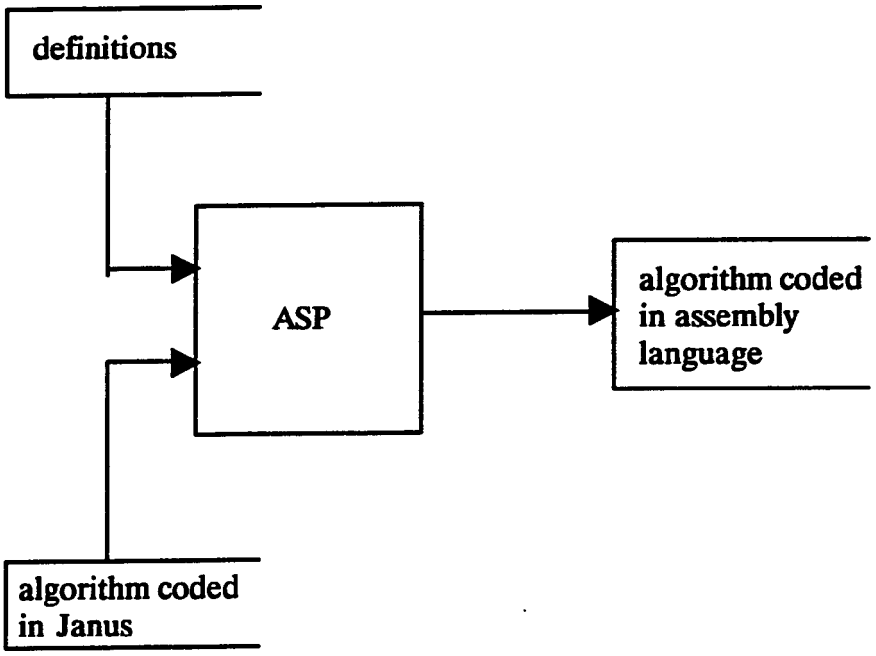


Figure 9.2. Data flow diagram of the translation process for Janus programs.

programming languages, and its contribution to portability as a target language for compilers [Waite, 1976, 1978].

9.2.2 Lispkit Lisp

Lispkit Lisp is a Lisp–like–language described at great length in Henderson [1980]. This work covers: the theoretical and practical issues in functional programming; a wide range of algorithms which can be implemented in a basic Lisp–like–language (Lispkit Lisp) and in an extended version of this language; a host of issues comparing and contrasting functional and iterative styles of programming; and last, but not least, a kit for implementing the purely functional language using a hypothetical language reminiscent of Pascal. A version of this latter piece of software has been implemented at the University of Alabama at Birmingham, in the C programming language, and has been used in classes and other studies [Dilworth, 1983].

A design for interaction between ASP and this compiler has been devised. In it, ASP creates "data" (in the form of code for the compiler), calls up the compiler, and receives results back from it. The design calls for a shared memory bank in main memory, a unit which has not been implemented, though its structure seems apparent. A prototype implementation has been effected using auxiliary files for data sharing.

CHAPTER 10

TAILORED LANGUAGES AND SYSTEMS

A tailored system in the Barrel/ASP context can be defined as a utility which provides a specific service for the user, such as text editing. In most cases, tailored systems are not complete systems but are tailored to fit specific applications. In other cases, tailored systems fall into the realm of an application program. We intend to use the term in a broad and sweeping manner to provide a context with which to describe many different kinds of implementations programmed through the facilities of ASP.

The distinction between kits and tailored systems is a fine one. In many cases it would be easy to classify a kit as a tailored system and vice versa. But a tailored system tends to be more than a kit or an implementation of language features. It tends to provide a narrower functional scope of service to the end user and in many cases can be used by a less sophisticated computer user.

The tailored systems we now describe are:

a) BLOGO, turtle graphics primitives from Logo,

b) BED, a text editor,

c) BQBE, relational database primitives,

d) BDT, a presentation processor,

e) BTINT, two presentation processors.

A listing of the statements included in each of these tailored systems can be found in Appendices 10.1 through 10.5.

BLOGO encompasses several of the turtle graphics primitives from the Logo programming language. Logo, essentially a dialect of LISP, is a simple but powerful

language developed for research in artificial intelligence [Abelson, 1982]. It is highly regarded as a means of introducing programming and problem solving skills to many different types of people – from handicapped children to computer science students.

Turtle graphics, a small part of Logo, is an implementation of turtle geometry where lines are described, not in terms of absolute position in a coordinate system, but relative to the position and direction of the turtle [Harvey, 1982]. A turtle is a conceptual animal (usually a small triangular pointer) that draws lines while moving around on the computer screen. It responds to a few simple commands. Some of the commands implemented in BLOGO are:

right	turn the turtle to the right
left	turn the turtle left
forward	move the turtle forward
back	move the turtle backwards
home	put the turtle in center of screen
cs	clear the screen
penup	make the turtle's pen inactive
pendown	make the turtle's pen ready to draw
hideturtle	make the turtle invisible
showturtle	make the turtle visible
repeat	repeatedly execute a list of commands
setbg	set the background color
setpc	set the pen color
setx	set the x coordinate
sety	set the y coordinate

Logo procedures can be implemented as ASP definitions. An example of a Logo procedure, the familiar POLYSPI program [Lawler, 1982], can be seen in Figure 10.1. It is implemented as an ASP definition and can be called, for example, by "polyspi 1 123 3", either interactively or in a program. The program displays different geometric designs based on the arguments provided.

```
polyspi # # #|
if #14 > 3*#24+#24/4 skip 3$
forward #14$
right #24$
polyspi #14+#34 #24 #34$
$
```

Figure 10.1. The familiar Logo procedure POLYSPI implemented as a definition for ASP.

Several points of interest about Logo are listed in Harvey [1982]; it is procedural, interactive, recursive, extensible, has list processing, and is not typed. The ASP environment compares favorably with each of these points and, in some ways enhances the Logo environment, for example, through the pattern matching capabilities of ASP which are lacking in Logo.

BED is a simple text editing facility. It can be used with other "kits" or tailored systems to provide editing without having to terminate the current Barrel/ASP session. Many of the standard commands found in most text line editors are implemented, including copy, delete, insert, substitute, find, undo, view, and list.

BQBE is an attempt to demonstrate the usefulness of pattern matching in manipulating small databases. Although small in scope (essentially an implementation of the project operation from relational databases), it is a potentially useful tool for many different applications. While still in the planning stages, one such application involves a decision table processor implemented in

terms of relational database operations. Since decision tables can be described in terms of a relational database [Salah, 1986] and other research work involving decision tables has already taken place (see Chapter 3) the implementation is easily conceptualized and complements the work already done.

BDT is an outgrowth of work which combines logic programming and decision tables. Logic programming was used to create "runnable specifications" [Davis, 1982; Kowalski, 1984a] for the implementation of a new decision table processor in Barrel/ASP. Specifications are runnable if they can be executed directly on a computer or a program can be generated automatically which is guaranteed to preserve the semantics of the specifications.

Prolog, a programming language based on first order logic (specifically Horn Clauses), has been shown to be adequate for use as a formal design language [Davis, 1982; Kowalski, 1984a; Moss, 1981; Warren, Pereira & Pereira, 1977]. In this case study, Bruynooghe's [1980] version of Prolog was used to devise runnable specifications for what we call a "presentation processor" for "procedures and regulations" decision tables [McDaniel, 1978]. This type of table is analogous to a menu in that it is user–oriented and deals with decisions, e.g., like those a manager might make concerning policy issues in an organization. Its contents thus can be presented to its users; they, in turn, respond and the system provides the appropriate course(s) of action to be taken. As an example, suppose we have a decision table consisting of actions to be taken based on the condition of certain types of cars. Figure 10.2 shows the dialogue BDT might use to correspond with the user. The responses of the user have been annotated with an asterisk at the beginning of the line. Chapter 11 gives a more complete description of presentation processors.

```
> % ccprolog input
  we are about to launch prolog-like action
  use -dt; for prompts
  use -det(cord,good); e.g. for direct entries
  send:
> -dt;
  car make is ?
> cord
  condition is ?
> good
  make is cord and condition is good
  commission is 1%
  shop work is 3 weeks
  manager ok is not required
  car make is ?
```

Figure 10.2. A typical dialogue of BDT with a user, annotated to indicate user input (by addition of >).

These tables are logically more complex than a menu, as one distinguishing feature. Another distinguishing feature, along a different axis, is that the table contents are complete in and of themselves and need not be translated by a preprocessor for subsequent compilation (usually) in a higher-level programming language.

Once the design was completed and tested a solution was effected with ASP as the implementation tool using existing Barrel/ASP kits. One of our goals was to make the Barrel/ASP solution as compatible with the Prolog specifications as possible. The rational for such a goal is the fact that both Prolog and ASP are based on pattern-directed computing. The closeness of the Prolog specifications to the

ASP implementation gives rise to the idea that ASP could be modified slightly to essentially become a Prolog–like processor.

A key point is that, though the logic program can serve as a presentation processor on a system which has a Prolog processor, our use of it as a specification suggests a method for implementing it through the pattern matching facilities of ASP. In this form, the presentation processor is made available to systems for which no logic programming is available, and in a manner that is based on logic expressed in a formal fashion.

BTINT is an implementation of two presentation processors similar to BDT. Rather than acting on a table which is built into the code (implemented in the Prolog version of BDT as clauses and as definitions in the Barrel/ASP version), BTINT acts on tables which are separate from the code and passed as data to the processors. The processors are part of a larger Barrel/ASP system which generates the tables to be interpreted by the presentation processors. Chapter 11 describes these processors fully.

CHAPTER 11

A TABLE ENTRY, REFORMATTING, TRANSLATION, AND PRESENTATION SYSTEM

11.1 INTRODUCTION TO PAR TABLES

Table processing software, written to be processed by ASP, centers on a view of table processing as a set of five related steps or phases which center around a combination of practical applications, formal definitions, and portability. The five steps are:

- creation, phase 1: code book creation
- creation, phase 2: rules entry
- reformatting
- translating
- presentation

It will be the task of the next several sections to clarify each of these phases. As part of the effort to do so, we shall use an example of a "procedures and regulations" (PAR) table [McDaniel, 1978]. Because such tables are meant for direct human use, with or without computer aid, they are easy to read and the processing they involve is almost apparent upon inspection. The table we use as an example, Figure 11.1, is also used as an example in Chapter 3 (Figure 3.1).

PAR tables are convenient for purposes of explication, but the reader should be aware that table processing involves a wider variety both of table forms and contents. There is one common thread, however, in that all tables represent an input–output correspondence. The correspondence is often described in terms of

conditions and actions though other correspondents are useful in some contexts, e.g., questions and answers or stimuli and response.

car make	cord	cord	reo	reo	duesenberg	duesenberg
condition	good	poor	good	poor	good	poor
commission	5%	1%	10%	5%	variable	variable
shop–work	no–need	3–weeks	no–need	3–weeks	6–weeks	6–weeks
manager–ok	no–req	no–req	no–req	no–req	req	req

Figure 11.1. Representation in table form of a decision procedure relating input values for "car make" and "car condition" to output values entitled "commission", "shop–work", and "manager–ok".

In the figure, each component of a six–part decision is represented in a column (to the right of "| |"). For example, in the table, a combination of input values such as reo and good (for car make and condition) maps into output values of 5%, no–need, and no–req, respectively (for commission, shop–work, and manager–ok). This table can also be read in an "IF ... THEN ... ELSE" fashion, e.g.,: IF (car make is reo AND condition is good) THEN (commission is 5% AND shop–work is 'not needed' AND manager–ok is 'not required').

Some writers call tables such as Figure 11.1 a "vertical, extended entry decision table." That such a table is called a decision table at first glance may seem strange since, put simply, it's just "a plain old table," but in fact, any input–output correspondence can be put into decision table format! Figure 11.1's vertical nature is apparent. The notion of "extended entry" is less easy to understand since the term describes the table's contents relative to a simpler form of table, a limited entry table, in which the contents of the cells to the right of the "| |" are restricted to yes or no (top half), and x (bottom half). The entries here, being words or simple phrases, are "extended" because they are not restricted to such simple forms.

Though this form of the table is quite easy to read (and use) by humans, it can be reformatted to make it easier and more efficient to process by computer. The present form also masks certain "good" properties that tables often possess. Among the latter are:

- completeness — all possible combinations of input are accounted for
- consistency — the table does not say, in one part, "Do A" for a set of conditions, and, in another part, "Do B" for the same set of conditions
- non-redundant — the same set of input and output do not appear in more than one place in the table

Some form of coding, therefore, is frequently imposed on tables, achieving a twin goal of making them easier to process by computer and also dramatizing the good properties. The rationale for having a "codebook" may be apparent in these remarks. Codebook entry is often a natural first phase of a table processing system.

The codes (of the codebook) can be used to facilitate the correspondences between the input and output items. Each such correspondence is called a rule; the second phase of a table processing system deals with rule entry.

PAR tables are a nice vehicle for discussing table processing because reformatting alone usually renders them ready for presentation. It is easy to envision the reformatting need by viewing Figure 11.1 and realizing that such information, particularly if the relevant contents are codified, can be put into a form which is much easier and efficient for a computer to process. As we shall see later, other kinds of tables require intervention of a distinct post-reformatting phase, a translation phase, almost always done by a "regular" compiler for some programming language such as Fortran or C.

A computerized presentation processor for a PAR table is generally an interpreter which prompts the user to provide input. Upon receiving it, the processor checks the table, matching the input with table contents, and reports the appropriate output to the user. We describe presentation processors in more detail in Section 11.2.5 below.

It is desirable to follow good systems analysis techniques when developing these tables and the processors which present them. Thus, we have integrated work previously done in the area of formal specifications for decision table systems (see Chapter 3) into this system of table creation and presentation. More specifically, it is possible to create "runnable specifications" for the decision tables and a particular presentation processor. The benefits of such specifications are well documented [Davis, 1982; Kowalski, 1984a] but if it were left up to the individual to generate them they would, more often than not, be left unattended. That is why, in the system described below, formal specifications are generated automatically each time a decision table is created.

11.2 BATERTAPS

One of the major applications written for Barrel/ASP is a decision table entry, reformatting, translation, and presentation system (the Barrel/ASP Table Entry, Reformatting, Translation, And Presentation System or BATERTAPS). The work was originally done as a masters thesis project at the University of Alabama at Birmingham by Charles Minderhout [Minderhout & Reilly, 1982]. I was involved in many of the technical decisions and made several significant enhancements after the initial work was completed. These enhancements include:

- the option of using graphics to enter the rules
- the expansion of phase 3 and the addition of phase 4 to allow for processing of tables expressed in a programming language

- the automatic generation of formal specifications

The system consists of five phases corresponding to each of the five phases mentioned above. The first two phases guide the user in creating a decision table. The result is a consistent coded extended entry decision table. Also, "runnable specifications" for the presentation processor described in Chapter 3 are automatically generated. The specifications are "runnable" because they are in the form of a Prolog program and can be executed on a computer which has a Prolog compiler. Prolog is a popular programming language based on first order predicate logic. The specifications are formal because Prolog has a well-defined fixpoint or denotational semantics as well as its proof theoretic semantics, which gives the specifications an adequate mathematical basis [Moss, 1981]. Thus, the user has a stable base to carry him or her through the next three phases. Or the last phases could be skipped entirely since the user already has an executable presentation processor.

The third phase of the system has three different components. Two of these components translate or reformat the information content of the table into a format which can be utilized by one of two presentation processors. The other component reformats the table so that it can be processed by a table processing system based on the C programming language. Phase four consists of the C table processing system while phase five consists of the two presentation processors. Tables are normally processed either by phase four or phase five but not both (however, different ideas are presented below).

One of the presentation processors was developed prior to the system, and one as part of the system, illustrating the adaptability of the approach. In addition, the fourth phase (translation) was added after the original work on the system was completed, further confirming the adaptability of the approach. Also, the second

phase has been enhanced by graphics; the user can elect to enter rules with the aid of an interactive display system (to be described in Chapter 12).

Although most of our work is with decision tables we do not limit ourselves to them. The presentation processors could be used for many types of tables. In this way we can generalize the concept of the tables and view them as a set of questions and answers. So any decision system in the form of a table can be processed by a presentation processor.

We also do not limit our presentation processors to tables for "procedures and regulations". Presentation processors can be used with the more traditional type of table coded in a high-level language. In particular, we envisage the design of a system which would facilitate the debugging of high-level language programs by allowing the user to provide the results of the conditional statements of the program and then displaying the actions taken by the program under those conditions. The availability of processors which translate programs to decision tables enhances the design of such a system. We could then use the reformatting/translation phase of the system to integrate communication between the various processors.

We also do not restrict ourselves to processing a single table at a time. The ability to process systems of tables linked together through constructs within the tables could easily be incorporated into the system. For example, one of the actions in a table could specify a new table to be processed. Instead of displaying the action the presentation processor could process the new table. We have, in fact, incorporated this capability in one of our presentation processors.

11.2.1 Phase I: Codebook Entry

Several authors stress that table processing needs to be user-oriented in all its phases [Metzner & Barnes, 1977; Montalbano, 1974]. The first phases, those of

creation of a table, may be most in need. That is, it is not expected that a user develop from scratch, in one step, a table such as that of Figure 11.1. Rather, parts of it are developed, e.g., within an interactive dialogue, perhaps using graphics aids.

Breaking the creation process up into phases allows the user to focus on smaller parts of the process, but also helps eliminate much clerical work on the part of the user. For example, conditions may be entered in near to natural language and the system can perform several editing chores for the user, supplying the codings for coded tables, displaying the table in different forms, and the like.

Figure 11.2 describes a portion of dialogue employed to create a table with the same logical contents as that in Figure 11.1. Note that when the dialogue is finished the user is permitted to continue whatever processing he or she is already engaged in. This includes possible editing through use of editors (such as the BED editor, see Chapter 10). The dialogue should be self-explanatory.

After suitable interaction, sufficient data may be obtained so that it is possible to display it conveniently as in Figure 11.3. Such a table is called a codebook. Each set of conditions and actions are put into an appropriate grouping, and a numerical code is supplied by the system to the options within these groupings. The codes are later used in entering rules for the table (see the rules entry processor described immediately below).

11.2.2 Phase II: Rules Entry

We also create rules for a table through interactive dialogue (see Figure 11.4). Rules connect the conditions to actions in the sense that the information content of Figure 11.3 is increased to contain the full table information exemplified by Figure 11.6 (and, content-wise, Figure 11.1). Note that this dialogue makes use of the codes generated in producing the code book; this facilitates the dialogue by abbreviating the information transfer from the user to the system.

```
                .
                .
  please
                a  "c" for entering conditions
                an "a" for entering actions
                an "e" to end input
>c
   please: a condition or "sample", "listc", "quit"
>sample
  make is [cord,reo,duesenberg]   ... is a sample to follow
>car make [cord,reo,duesenberg]
   please: a condition or "sample", "listc", "quit"
>condition [good,poor]
   please: a condition or "sample", "listc", "quit"
>listc
 car make [cord,reo,duesenberg]
  condition [good,poor]
   please: a condition or "sample", "listc", "quit"
>quit
  please
                a  "c" for entering conditions
                an "a" for entering actions
                an "e" to end input
>a
   please: an action or "sample", "lista", "quit"
>sample
  comm is [1%,5%,10%,variable]    ... is a sample to follow
>commission [1%,5%,10%,variable]
                .
                .
                .
   please: an action or "sample", "lista", "quit"
>quit
  please
                a  "c" for entering conditions
                an "a" for entering actions
                an "e" to end input
>e
  processing ....
  to screen the table ... enter:  sc
>sc
                .
                .
(A display like that of Figure 11.3 appears here.)
                .
                .
  to diskout the table ... enter:  do
>do
   table about to go to disk
  the external name of your table is : classic
  the internal name for the table is tab
  we are now in interactive mode ... please have fun
> stop
```

Figure 11.2. Portion of the codebook entry processor dialogue, annotated to indicate user input (by addition of >).

car make	1:cord
	2:reo
	3:duesenberg
condition	1:good
	2:poor
commission	1:1%
	2:5%
	3:10%
	4:variable
shop–work	1:no–need
	2:3–weeks
	3:6–weeks
manager–ok	1:req
	2:no–req

Figure 11.3. Result of the codebook entry portion of the system: a set of conditions (top half) and actions (bottom half) for a simple "procedures and regulations" table with appropriate system generated codes (e.g., 1 for cord, 2 for reo, etc.).

The table created from the dialogue in Figure 11.4 is available in the form of Figure 11.6. During entry, the user can consult information in the format of Figure 11.5, which together with the codebook provides guidance in entering rules (as is seen in Figure 11.4).

The system does not allow the user to duplicate the conditions, so that it is impossible in the dialogue to enter either inconsistent or redundant rules. Thus, once the rules have been entered the rules entry processor can create a consistent coded extended entry decision table such as that of Figure 11.6.

The user is given an opportunity to choose to enter rules either in a text oriented format or to enter them aided by graphics. In the figure, the user elected to enter data in textual format. Had he chosen the graphics option, a different form of dialogue ensues (see Chapter 12).

```
  to diskin the table ... enter:  di
>di
  enter name of codebook for rules to be generated
>classic
  do you wish to use gigi graphics to enter the rules?
>n
  please enter rules in this form: 3 2 : 4 3 2
  to exit please enter "quit"
  please a rule like "3 2 : 4 3 2 " or "cbook" or "rules" or "quit"
>1 1 : 1 1 2
  please a rule like "3 2 : 4 3 2 " or "cbook" or "rules" or "quit"
>cbook
                .
(A display of Figure 11.3 appears here, for user consultation.)
                .
                .
  please a rule like "3 2 : 4 3 2 " or "cbook" or "rules" or "quit"
>rules
                .
(A display like that of Figure 11.5 appears here.)
                .
                .
  please a rule like "3 2 : 4 3 2 " or "cbook" or "rules" or "quit"
>quit
  processing ....
  to screen the extended-entry table ... enter:  sc
>sc
                .
(A display like that of Figure 11.6 appears here.)
                .
                .
  to diskout the extended-entry table ... enter:  do
>do
  table about to go to disk
  the external name of your codebook is : classic
  file name for table ?
>classicdt
  to diskout the rules ... enter: do
>do
  file name for rules ?
>classicru
  do you want to generate formal specifications for your table?
>y
  processing ...
  to screen the formal specifications ... enter:  sc
>sc
```

Figure 11.4. Portion of the rules entry processor dialogue, annotated to indicate user input (by addition of >).

```
.
.
(A display like that of Figure 11.7 appears here.)
.
.
  to diskout the formal specifications ... enter:  do
>do
  file name for specifications ?
>classicspecs
```

Figure 11.4. (continued)

```
1 1 : 1 1 2
1 2 : 2 2 2
2 1 : 2 1 2
2 2 : 3 2 2
3 1 : 4 3 1
3 2 : 4 3 1
```

Figure 11.5. The rules table created by the rules entry portion of the system with the numbers representing the codes from the codebook.

```
car make          * 1 1 2 2 3 3
condition         * 1 2 1 2 1 2
**********************************
commission        * 1 2 2 3 4 4
shop-work         * 1 2 1 2 3 3
manager-ok        * 2 2 2 2 1 1
```

Figure 11.6. Result of the rules entry portion of the system (a consistent coded extended entry decision table) with the numbers representing the codes from the codebook.

Once the complete table has been created the user has the option of generating formal specifications for the decision table. Actually, the specifications are for a particular presentation processor but the table is hard coded into the processor. Thus, we generate different specifications for each table. The specifications produced for this example are seen in Figure 11.7.

```
dt :- write('car make ? '), read(C1),
      write('condition ? '), read(C2),
       intermed(C1,C2).
intermed(no,C2) :- write('so long now'), nl.
intermed(C1,C2) :- dec(C1,C2), nl,
                        write('we continue'), nl,
                         dt.
dec(C1,C2) :- table(C1,C2,A1,A2,A3),
                write('commission '), write(A1), nl,
                write('shop-work '), write(A2), nl,
                write('manager-ok '), write(A3), nl, nl.
dec(C1,C2) :- not(table(C1,C2,A1,A2,A3)),
                write('input values not found in table'), nl.
table('cord','good','1%','no-need','no-req').
table('cord','poor','5%','3-weeks','no-req').
table('reo','good','5%','no-need','no-req').
table('reo','poor','10%','3-weeks','no-req').
table('duesenberg','good','variable','6-weeks','req').
table('duesenberg','poor','variable','6-weeks','req').
```

Figure 11.7. The formal "runnable specifications" for the presentation processor which operates on the example decision table.

11.2.3 Phase III: Table Reformatting Processors

Once the rules have been added, i.e., a complete table is obtained, there are many other forms of processing that can be done. For example, tables can be reformatted into a variety of forms, such as condition policy maps, action policy maps, and even described in narrative form [Montalbano, 1974].

We have developed several reformatting processors to prepare tables for direct display of table contents on an interactive basis or to prepare tables for translation to procedural code. Our first reformatting work was for two interactive presentation processors discussed in Section 11.2.5. In a later example,

reformatting is done to provide input to a decision table processor based on the C programming language.

11.2.3.1 Reformatting for Presentation

Figure 11.8 shows the example PAR table after it has been reformatted for input into one of the presentation processors. The format at the beginning of the table is very similar to the codebook format. Then comes the number 6 which is the number of rules. This is followed by 6 pairs of condition entries (utilizing codes from the codebook) and action entries (on the next line). Each pair makes up a rule from the table. The condition entries specify an option from each condition group, for example, "1,1," means "car make: cord; condition: good". The action entries specify all the actions to be taken for the conditions. The actions are numbered sequentially, so, for example, "1,5,9" means "commission: 1%, shop–work: no–need, manager–ok: no–req".

11.2.3.2 Reformatting for Translation

In order to show the versatility of this approach to table processing we decided not to restrict ourselves to tables of the PAR variety but to allow processing of tables which are expressed in terms of a programming language [Humby, 1973]. There is a lot of software available to the public for translating such tables to code that a compiler can handle.

We have several such translators available to us to choose from for inclusion in our system (see Chapter 4). The one we chose is based on the C programming language. It accepts a combination of C programming language code and decision tables in a particular format with "C–like" conditions and actions and it produces C programming language code. The system is called DELTRANS and was developed by Keller and Roesch [1977].

```
do
car make          1:cord
                  2:reo
                  3:duesenberg
condition         1:good
                  2:poor
====================
commission        1:1%
commission        2:5%
commission        3:10%
commission        4:variable
shop-work         1:no-need
shop-work         2:3-weeks
shop-work         3:6-weeks
manager-ok        1:req
manager-ok        2:no-req
6
1,1,
1,5,9
1,2,
2,6,9
2,1,
2,5,9
2,2,
3,6,9
3,1,
4,7,8
3,2,
4,7,8
```

Figure 11.8. Results of Phase III, the reformatting phase. The example "procedures and regulations" table is now ready for processing by a presentation processor.

The first two phases for developing such a table are the same as for PAR tables. The user simply follows several different conventions for entering the conditions, actions, and rules.

Figure 11.9 shows the output of the codebook phase for a sample table. Note that the conditions are binary; i.e., they are either true or false. Thus, the groupings so prevalent in the PAR tables are not necessary. The conditions are entered with empty groupings, e.g., "r < 0 []". The actions are also entered with empty groupings since they, likewise, do not lend themselves to be easily grouped. Of course, we could have chosen to enter y and n in the brackets (e.g. "r < 0 [y,n]"). While being perhaps more readable, that convention was deemed too burdensome for the user.

```
r < 0                       1:
s < 0                       1:
t < 0                       1:
====================
printf("T < 0 : ")          1:
printf("S < 0 : ")          1:
printf("R < 0 : ")          1:
rpos()                      1:
```

Figure 11.9. A sample codebook with conditions and actions derived from statements from the C programming language. The table is being prepared for processing by the DELTRANS table processing system.

When the rules are created the user enters a 1 or 0 instead of codes from the codebook. A 1 in the condition portion of a rule means yes or true; a 0 means no or false. Dashes ("–") are also allowed in the condition portion of the rules and are used to signify "don't care"; i.e., it does not matter whether this condition is true or false. A 1 in the action portion of a rule means take this action while a 0 means do not take this action. After all the rules are entered a limited entry decision table

```
r < 0                                   * 1 1 1 1 0
s < 0                                   * 1 1 0 0 -
t < 0                                   * 1 0 0 1 -
************************************************
printf("T < 0 : ")                      * 1 0 0 1 0
printf("S < 0 : ")                      * 1 1 0 0 0
printf("R < 0 : ")                      * 1 1 1 1 0
rpos()                                  * 0 0 0 0 1
```

Figure 11.10. A sample decision table with conditions and actions oriented towards the C programming language. It is ready to be reformatted for use in the DELTRANS table processing system.

(Figure 11.10) is produced. That table is then ready to go through the reformatting phase.

Figure 11.11 shows the dialogue from a typical session with the reformatting processor. The user can enter a "label" for the table which is typically used to provide information which would allow the resulting C code to be used as a function or subroutine. The user can also enter declarations for any variables that are used in the table. The resulting table is ready for processing by the DELTRANS processor.

11.2.4 Phase IV: Table Translation Processors

In most cases, this phase involves translating tables to procedural code in a specific programming language and then compiling the code to produce an executable program. These two steps may or may not be separate. Unfortunately, most compilers are not well integrated with table processing, so that little error detection and correction is possible once a table is translated to procedural code. Thus, the creation and debugging phases are distinct – a situation which produces neither the shortest system lifecycle nor the most dependable code. We have

```
  Phase 3 – Reformat Decision Table for C Translation Processor
  to diskin the table ... enter: di
>di
  enter name of table
  You have asked to use a new file (file number 7).
  Type in its name please.
>tdt
  processing ....
  what label do you wish to give this table?
  e.g. "tab(c)  int c; { "  if it is to be a subroutine
  hit return if you do not wish to label it
>tab(r,s,t)    int r,s,t; {
  any declarations?
>n
  to screen the table ... enter:  sc
>sc
  <
  n tab(r,s,t)    int r,s,t; {
  r 5 c 3 a 4 e @
  ^
  r < 0              @y(1)y(2)y(3)y(4)n(5);
  s < 0              @y(1)y(2)n(3)n(4)–(5);
  t < 0              @y(1)n(2)n(3)y(4)–(5);
  ^
  printf("T < 0 : ") @1,4;
  printf("S < 0 : ") @1,2;
  printf("R < 0 : ") @1,2,3,4;
  rpos()             @5;
  <
  to diskout the table ... enter:  do
>do
  table about to go to disk
  file name for table?
>ctable
```

Figure 11.11. Portion of the reformatting processor dialogue for tables to be entered in the DELTRANS table processing system, annotated to indicate user input (by addition of >).

already mentioned (in the introduction to this chapter) the possibility of using a presentation processor to provide a debugging facility for these tables.

Tables which contain programming language code or even employ a particular syntax as a concession to facilitate computer translations are generally less readable than PAR tables, since knowledge of the programming language for which they are coded is a virtual necessity. Also, the consequences of executing them are frequently less apparent, e.g., if they require a value to be read in from an external device or computed, say, using a random number generator.

Display of table input and output, perhaps with simulation of computations, is possible in such cases, and is, in fact, considered by us briefly (specifically, in the introduction to this chapter). Sometimes, however, such tables are meant to be processed in batch mode, a reason for compiling instead of interpreting being to attain maximum efficiency. We may still refer to a presentation processor under these circumstances, though the input that is presented to the compiled code generated from the tables may come from sources other than on-line users, e.g., from external files or from other procedures, and the presentation phase usually consists of the execution of the program of which the table is a part. For most table processing systems, the calling procedures may or may not have been derived from tables.

It is important to recognize that by having independent reformatting and translation components, we can interface our table construction methods to a variety of table processing systems. This includes ones which we obtain from other workers, the presence of which contributes greatly to the portability and networking potential for our table systems. We have illustrated this point in a previous chapter (Chapter 4), wherein we list several "imported" processing systems along with some we have developed.

As a matter of fact, we have not written any code for this phase but rely on "off the shelf" software. Specifically, we use the DELTRANS table processing system which we have already described in the section on reformatting (Section 11.2.3.2).

11.2.5 Phase V: Presentation Processors

In the previous sections we have shown how a user can formulate a table. We now describe one way this information can be used. The table's conditions (questions) are displayed (presented) to the user; the user selects an appropriate option; and the actions (answers) are presented to the user. Again, procedures and regulations (PAR) tables are most appropriate for our purposes, so we continue with the example discussed above.

We include two presentation processors in our system. Our first one, which processes only limited entry decision tables, was implemented on the Data General in the macro assembly language MASM, prior to any of the work on table entry. The same processor was later implemented in the Janus language (see Section 9.2.1, Chapter 9). Still later, this processor was implemented in Barrel/ASP code.

For tables to be processed by this presentation processor, the reformatting phase must reformat the table from a coded, extended entry table to a limited entry table. Thus, the input by the user is limited to 1s and 0s to answer yes or no to the conditions. A small display of the interaction suffices; see Figure 11.12.

Our second presentation processor, which processes coded, extended entry tables, takes more advantage of the codebook format of the table. Each condition group is displayed along with the the codes and the user enters the appropriate code. A sample of the dialogue is seen in Figure 11.13.

Some of our presentation processors (but not the ones included in this system) have the capability to process systems of tables. In them, the action portion of the table transfers control to another table for further processing. In principle, the

```
  to continue: y or yes
>y
  1 for yes ... and 0 for no
  car make cord
>0
  car make reo
>1
  car make duesenberg
>0
  condition good
>1
  condition poor
>0
  commission 5%
  shop–work no–need
 manager–ok no–req
```

Figure 11.12. Partial dialogue from execution of a typical presentation processor, annotated to indicate user input (by addition of >). The last three lines are the actions displayed by the system.

condition portion of tables also can invoke other tables, just as they might invoke arbitrary functions. Minderhout and Reilly [1982] and Salah, Reilly and Yang [1984] discuss such systems.

```
  car make          1:cord
                    2:reo
                    3:duesenberg
  type the appropriate number
>2
  condition         1:good
                    2:poor
  type the appropriate number
>1
  the actions for this case are:
  commission        2:5%
  shop-work         1:no-need
 manager-ok         2:no-req
```

Figure 11.13. Partial dialogue from execution of a typical presentation processor, annotated to indicate user input (by addition of >). The last three lines are the actions displayed by the system.

CHAPTER 12

GRAPHICS CREATION AND EDITING OF TABLES

Graphics are a necessary component in a modern decision table system. Decision tables are graphically oriented because they are inherently two-dimensional. The focus of this chapter is on using graphics to display tables, especially during the creation phase of table processing as described within the Barrel/ASP context (see Chapter 11).

Graphics are used in conjunction with the Barrel/ASP Table Entry, Reformatting, Translation, and Presentation System (described in Chapter 11) in several ways:

- entering the rules of the table (during Phase II),
- limited editing of the rules after they have been created,
- and to display the graphical representation of the table any time after it has been created.

The codebook created in phase I is used to create an empty limited entry decision table (i.e., one with no rules). This table is displayed on the screen along with one as yet empty rule. The cursor is positioned over each condition one at a time and a response is elicited. The condition being processed is highlighted. The user types a 'Y' if the condition is true or anything else if it is not. A 'Y' or blank is displayed for that condition. The actions are treated similarly, using an 'X' instead of a 'Y'. If an inconsistent or redundant rule is entered an error message is displayed and the rule is not accepted. Figure 12.1 shows the example table used for illustration in Chapter 11 (Figure 11.1) in the process of being created via the

graphics editor. The 5th rule is being entered and the "shop–work is 6–weeks" action is highlighted, awaiting a response by the user.

car make is cord?	y	y			
car make is reo?			y	y	
car make is duesenberg?					y
condition is good?	y		y		y
condition is poor?		y		y	
commission is 1%		x			
commission is 5%	x			x	
commission is 10%			x		
commission is variable					x
shop–work is no–need	x		x		
shop–work is 3–weeks		x		x	
shop–work is 6–weeks					
manager–ok is no–req	x	x	x	x	
manager–ok is req					

Figure 12.1. Entry of rules via graphics editor.

When a rule has been completed the system asks if there are any more rules to be entered. If the answer is yes then another empty rule is generated and the process begins again. Otherwise the user is asked if there are any corrections to be made. If the answer is yes the system asks which rule number needs correcting. The cursor is then placed at the beginning of that rule and the user fills it in again. When all corrections have been made a coded, extended entry decision table is produced (just as if the user had chosen to use the non–graphic method of entering the rules).

Two of the extensions made to Stage2 which gave rise to the Barrel/ASP processor (see Chapter 6) are utilized in the graphics approach to rules entry.

Most definitions necessary for the Barrel/ASP processor to run are entered at the time the Barrel/ASP processor is started. But the definitions required for the graphics processing are not entered until the user decides which mode of rules entry is desired (graphics or non-graphics). The processor function for adding new definitions at any time during processing is used to add the graphics definitions. The second new extension used is the processor function which provides the interface to the graphics terminal and allows graphics commands to be used.

The user has the option of saving all of the graphics commands used in generating the table. The commands are saved in a file chosen by the user. At any time later, the user merely has to execute that file of Barrel/ASP statements to get a graphics display of the table on the terminal.

CHAPTER 13

SUMMARY

This thesis has dealt with a proposed methodology for software development, using computerized, formal techniques, in a context where the software is to be developed with specific development tools. These tools are ones which we have devised and shown to be flexible, portable, extensible, and easy to use. They were designed with a targeted goal of providing a formal specifications methodology and implementation tools for persons engaged in simulation, particularly as it relates to newest forms of simulation involving non–numeric computation.

Each of these three elements, the formal techniques, the software development tools, and the applications area, have received considerable attention in this document. Extensive illustrations and theoretical contributions have positioned us to make recommendations about formal definition techniques at the detailed, practical level required to make this methodology acceptable to simulators. The work enables us to provide convincing evidence that developing (simulation) applications with a high degree of formal methods is not just desirable from a computer science perspective but is practical as well.

The formal techniques are based specifically on logic programming and Prolog. A dual level, complementary approach, i.e., both at the programming statement level and at a "higher" systems (components) level has been presented, justified, and demonstrated.

The software tools we developed for implementation, the Barrel and ASP processors, complement the theoretical work so that a variety of functioning

software has been shown to be under control of both theory and new and powerful software development tools.

The applications area is served through practical demonstrations of both the formal techniques and the software development tools, with examples that are picked for potential interest to simulators. These provide an adequate test of the effectiveness of the proposed methodology in the applications area. With these summary remarks in mind we now review each chapter of the dissertation and summarize its relationship to the thesis.

13.1 PART 1

After an introductory chapter we described, in Chapters 2 through 4, a computerized formal methodology, which we feel can have a significant impact on software development in simulation modeling. This methodology allows us to reason about the components of models as they are being built, and, thus, contributes to the development of simulation elements, such as sophisticated E–units and other portions of the BEAK environment (explained further in section 6 of Chapter 14).

One of our major objectives has been to identify appropriate tools and associated programming language and system theory to support our view. The criteria we used in locating potential candidates included:

- executable on a computer
- extensible
- flexible
- logic based

We required that the formal techniques be capable of being executed on a computer. Otherwise, they would be essentially useless to supporting the ultimate goal of automating the entire BEAK.

The methods also had to be sufficiently adaptable to change, with respect to both extensibility and flexibility. We expect that non–numerical computing, in all of the BEAK units, is now only partially understood. Accordingly, we feel that a tool that is unable to respond to change would be next to useless. Additionally, its implementation must be flexible and not burdensome when it is moved to new machines and machine environments.

We chose to base our formalizations on (formal) logic. Translated into today's terms, this means some form of logic programming. We chose, in addition, to try to stay as close to Horn Clause forms as possible so that we could make extensive use of Prolog. This is important not only because of the strategic relationship of Prolog to symbolic computing, but because the software we have been developing can itself emulate Prolog.

Chapter 2 described our formal techniques at the programming statement level. We demonstrated how non–numeric simulation language constructs can be developed through the use of formal techniques based on Metamorphosis Grammars. A language we defined, Barrel–F, was designed to include several essential features for symbolic simulation; among them are table processing and string manipulation.

We demonstrated the extensibility of the method by showing how the definition can be extended as the language is extended. We provided an example by adding stack and queue manipulation statements to the language and discussing the steps required for extending the definition.

We then used the formal definitions to implement the language through the use of our non–numeric software development tool, ASP. We provided a suite of test programs for testing both the definition and the implementation.

We also provided a formal definition of the development tool, ASP. We defined much of the process of writing code bodies in ASP, including most of the parameter transformations and processor functions. We believe this to be a unique case for this kind of software, though it has obvious traces to proving operating systems and compilers correct.

In Chapter 3 we moved our formal techniques to a higher level of processing, that of programming systems. These are larger units of code, often standing alone, and in some particulars bear resemblance to the "object" of object–oriented programming. We did not require this object oriented discipline in the current implementations, but it appears sufficiently related that a follow–up study could be launched to develop it (see Chapter 14, section 2).

We used Prolog to provide "runnable specifications" for a decision table "presentation processor," a table processing system component. We then implemented the processor using ASP. The flexibility of ASP was demonstrated by making the implementation correspond as closely as possible to the (Prolog) specification. We discussed some of the differences between Prolog and ASP and how some features which are natural in a simplest formulation of a table processor in Prolog (for example, input/output indifference) might be addressed in ASP. We also introduced the idea of automatically producing the specifications through interactive user input of the requirements for the table.

In Chapter 4 we showed how our desire for portable systems led to the introduction of a portability concept we call "portability–cubed." We described three instances of portability that can be used separately or together in software development work.

The first instance involved the portability of the ASP processor itself. ASP uses an abstract machine approach to implementation and can be easily ported to

other systems. We described the full bootstrap method of implementation as well as the three ports we have done.

The second instance revolves around the ability to use ASP to process languages which mimic "real" programming languages. In this way, applications written with ASP can be ported to systems which support the language.

The third instance involves a network of table processing systems. Tables generated by ASP table processors can be manipulated by several different table processors located on various machines. The scorecard: about nine implementations on about five machines.

13.2 PART 2

Chapter 5 began our introduction to the Augmented Stage2 Processor, ASP. Here we described what ASP is, what it is used for and how it works. We discussed its origins in William Waite's Stage2, its identity as a general purpose macro processor, and its ability to do string transformation as well as programming language interpretation. We talked about using it to analyze programming language features through synthesis of these features, to study different modes of processing (functional, logical, procedural), and to prototype and implement systems. We talked about its place in the formal studies and its place in the simulation environment. We provided a tutorial on how ASP works – macro definitions, templates, code bodies, calls, parameter transformations, processor functions and associative memory.

Chapter 6 described the background of ASP (i.e., Stage2, the tool it is based on). We described the extensions and modifications made to Stage2 to arrive at ASP as well as the methods employed in making the modifications, principal ones being extending the abstract machine and adding to embedded code.

The abstract machine approach to the implementation of ASP, as well as the various machines and programming languages used in actual implementations, were described in Chapter 7.

13.3 PART 3

Chapters 8 through 12 described various applications developed to demonstrate the usefulness of ASP in developing symbolic software. We have adopted a name for these applications and demonstrations; we called them the Barrel system.

Chapter 8 showed how we were able to use ASP to analyze various programming language features which we deem useful for symbolic processing. The focus at this point was on individual statements, some of which were proposed in journals for inclusion in modern programming languages; others being derived from existing languages which appear to have a place in symbolic simulation. Several such implemented statement types were used in various contexts and environments.

Chapter 9 described the various "kits" or collections of related programming language features developed through the use of ASP. These kits contain many of the types of features which we described as necessary for symbolic processing; inspiration from Lisp, Snobol, and other more "conventional" features are apparent. We discussed how these kits can be used separately or in conjunction with other kits.

Chapter 10 focused on the portions of Barrel which are tailored toward a specific service or application. These "tailored systems" are similar to kits but usually are more limited in scope. Examples include a text editor, turtle graphics and table "presentation processors."

Chapter 11 described a particular table processing system which is capable of asking for requirements for a decision table and, in a series of five phases, producing decision tables in various forms for processing by various (other) decision table processors. The system is also capable of automatically producing runnable specifications for a table processor of the procedure and regulations type, as discussed in Chapter 5.

Finally, Chapter 12 talked about the capability of using graphics to provide the input to the table processing system of Chapter 11.

Thus, the three parts of the dissertation delineate the thesis statement in microcosm: Software for simulation systems should be developed within a framework which has both a strong theoretical foundation (Part 1) and a useful and practical application package (Part 2). Furthermore, to be truly relevant to the needs of the users, a computerized methodology is preferred over a manual methodology, and the implementation tool must be extensible, portable, and easy to use. A sufficient basis for a formal methodology for simulation modeling in a broadened sense of combined continuous, discrete and symbolic simulation is a methodology we have developed in logic programming (Part 1); an implementation tool that satisfies the stated needs for developing software for simulation models and environments, with primary emphasis on the symbolic and non-numeric areas, is the system we have developed, which includes a base processor (Part 2), a core facility forming a first-cut symbolic simulation language (Part 1), and much associated utility code (Part 3).

CHAPTER 14

THE FUTURE

The present work has positioned us for a number of interesting issues and opportunities. After brief remarks on several of them, we conclude with a larger tract on the BEAK simulation environment opportunities.

14.1 AN INTEGRATED IMPLEMENTATION OF LOGIC METHODOLOGIES

The two principal components of the current methodology, the language level and the programming systems level, have been integrated only in a conceptual way; each of these component levels is separately employed in demonstrations. What is needed for future work is a seamless garment wherein shifts from one subsystem to the other can be made automatically, even without user intervention. The two components could then be merged into a single comprehensive one, the processor depicted in Figure 14.1 describing at top-level a design whereby this can be effected.

In this design, the processor receives program descriptions (usually code in sequential language style, C or Fortran being typical cases) and analyzes them syntactically and semantically. Besides this, however, the processor also accepts runnable specifications, in Prolog, and processes them, just as it would C or Fortran. The part that processes Prolog statements is immediately implementable by adopting some code from Christopher Moss' thesis. It can join with the code we have developed, in providing full coverage of all we have designed for ASP and Barrel.

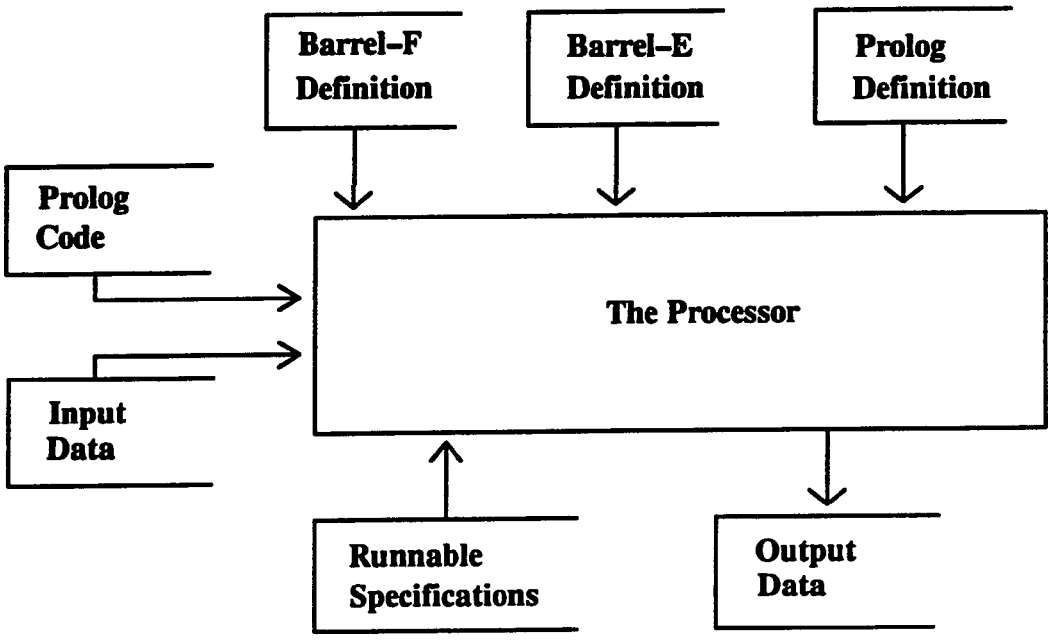


Figure 14.1. The combination of programming language definitions with programming system definitions.

Not to be forgotten is that it may be possible to build from the foundation established in this thesis, wherein the processor automatically translates logic specifications into Barrel code. So far we have dealt with examples which we feel provide feasibility information. The next step is to proceed to a more systematic frame. In so far as this is possible, only the statement level system, which we have covered comprehensively, would be needed.

If both a (Barrel) statement and a (Prolog runnable specifications) module level component are involved in the (future) system, some mode switching mechanism needs to be designed to allow switching between the alternatives. In crudest form this mechanism would require the user to proffer a signal, e.g., some character or pair of characters.

More interestingly, the system may be able to figure out on its own what is the nature of the code it is receiving. This is, it seems apparent, a non-trivial problem if

Prolog–mimicking Barrel code is envisioned along with Prolog specifications. Accordingly, we leave the problem to follow–up work.

Another set of complications would occur if a higher form of logic programming, or a specialized logic is introduced in new components. These, too, create a potential for detailed and comprehensive additional work suitable for doctoral level research.

14.2 OBJECT–ORIENTED ANALYSIS AND IMPLEMENTATION

There is room for improvements in another area we mentioned earlier – taking a more formalized view of objects and notions that go with the object–oriented paradigm. We have informally analyzed some of the basic concepts and believe they would not, in general, be difficult to impose. Object–oriented thinking is making a large impact in simulation, our primary domain of interest, and any effort in increased emphasis on object–oriented philosophy would be welcomed on two counts: the underlying methodology *and* the applications domain.

14.3 LINK TO THREADED LIST THEORY AND PRACTICE

Another opportunity relates to "threaded list" languages such as Forth. That Barrel/ASP can mimic Forth, in principle, i.e., at a capabilities level, has really been amply proved in the present dissertation. That is, we actually use the core idea of statements made to the processor (ASP machine language statements) causing execution of pre–compiled function– or routine–level codes. This is done extensively in the graphics drivers and the Logo graphics mimicking actions. The basic call mechanisms in Forth are essentially a special case of the mechanisms employed in ASP.

A problem worth consideration relative to Forth comparisons is efficiency. Forth is tuned to efficiency, in part, by the very same restriction we mentioned in the previous paragraph. An obvious but not elegant mechanism, within ASP, would be to use flags which identify modes: a "Forth mode" and a "non–Forth mode." This could be done on an individual statement basis or at the level of blocks of code. Possibly, in some implementations (see comments below on parallel implementation), the system could employ a "Forth mode first" or a "Forth mode in parallel" scheme.

We note once again that research and development into these matters would carry a pay–off for simulation, probably most so in areas where models are integrated with instrument and measurement devices. An example might be monitoring of computer–communications networks using a programmable set–up like that surrounding a UAB CIS system which has an embedded data–collecting computer.

14.4 PARALLEL AND DISTRIBUTED PROCESSING

In the bulk of our previous work there has been a bias toward mini– and micro–computers. But, more recently, the advent of parallel machines, such as the Sequent here at the UAB, and access to the Alabama state supercomputer, has forced us to address a wider variety of computers in a minority of our studies. We have already mentioned parallelisms in our putative Forth mode, but there are many additional possibilities; we explore a couple of them here.

A future is now envisioned for some UAB research on distributed simulation, coordinating with other members of the CIS staff, e.g., on networks distributed over ethernet and networks of supercomputers connected by hyper–channels. These newer systems are often exclusively or predominantly Unix systems. Our past work has often been in Unix–based systems (often Vaxen, and in one our most recent

studies, the Sequent Balance), though we are in no way limited to such, having developed systems under the Data General AOS operating system. Specifically, then, choice of programming languages and approaches to Unix based environments are influenced by needs for portability and flexibility.

Parallel and distributed connections seem to present a number of opportunities. The basic tree searches used in ASP can be parallelized. In applying the pattern-directed computation style to very large problems, we can isolate "vocabularies" to different processors and let competition reign among them to decide winners. One of our earliest efforts involved a processing style of this type, in a study designed to gain efficiencies in memory management. We distributed ASP's tree searches over separate processes [Barrett, 1981b] and achieved some success in the goal, but found other issues we would need to solve to achieve full success. As a result of reflecting on these matters, we could see readily that research and development into tasking and parallel processing is a deep study with matters such as conflict resolution and ultimate efficiency for very large problems being among focuses.

14.5 ADJOINING CURRENT WORK WITH PREVIOUS LOGIC THEORY

Reilly, Salah, and Yang [1987] explored several term-predicate relationships for decision tables (DTs), assessing properties and then showing that, mostly because of input-output indifference, a form used often is very useful: table (cord, good, 10%, 2_wks, no-req.). In this case, all terms are constants. Cases with more complexity include ones where functions are allowed in generating output values; these put some restrictions on the extent of input-output indifference.

The first of the two just mentioned forms has charm with respect to storing it in relational databases, since it is just that, a relation. We get some help from a

processor which implements the restrictions of relational database, e.g., in the form of eliminating duplicate or conflicting entries, since relational databases do not allow duplicate keys. However, in practice, there are some limitations: we may not *always* want to eliminate duplicates. For example, we may wish to keep an old rule around until we break in a new rule. Both Prolog, sans the database option, and Barrel allow us this broader mode of operation. Reconciling these modes of operation suggests at least some practical matters to investigate.

Other forms of representation discussed by Reilly et al. [1987] include what Salah calls "an implicational form." The epitome is: input(cord,good) <— output(10% ...etc). Such a form provides a facile way to produce a better match between a rule-based system statement specification in Prolog and the implementation we presented earlier in Barrel. This improvement is balanced by the loss of input-output indifference, unless we explicitly invoke metaprogramming in the Prolog case.

A "functional form" (e.g., input([cord, good])) was also presented by Reilly et al. [1987]. An important point was made that any form which places a function in the path of automatic Prolog access (automatic to predicates) puts a restriction on the generality of DT that can be processed. Barrel seems able to step around these limits, albeit, again, through what amounts to metaprogramming (when viewed through Prolog colored eyeglasses). More exploration is called for at this point, especially, in any serious attempt to bring the Reilly and Salah schemes into the service of the simulation environment.

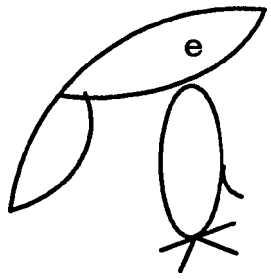
There are two specific areas for which Barrel and ASP seem particularly well suited for integration with the work of Salah, Reilly and Yang. One is to provide for automatic conversion from one DT representation to another. The second involves a particular type of DT rule problem where the input to the table matches if "any k

of n" parts of the rule matches. Salah and Reilly [1987] covered this kind of rule; their algorithm seems to present no difficulties for programming in Barrel and could be a nice addition for use in certain studies of simulation.

14.6 SIMULATION ENVIRONMENTS AS THE "BEAK"

The top–level activities in a simulation environment can be expressed by the acronym BEAK, an expression coined by Reilly and used by him and his colleagues, including us, in several recent papers [Reilly et al., 1984; Reilly et al., 1985; Reilly et al., 1986; Reilly & Dey, 1987; Reilly, Jones & Dey, 1985]. BEAK stands for:

- **B : Build**
- **E : Execute**
- **A : Analyze**
- **K : Knowledge**



In simplest terms, a simulationist: 1) *builds* models, 2) *executes* them, 3) *analyzes* their results and 4) contributes appropriate results to a *knowledge* base.

In more complex form, subcycles may appear within the BEAK. A very familiar one is that operating between the analysis routines of the A–unit and the knowledge base of the K–unit. The user, in this subcycle, is in the act of exploiting results of a model whose run (E–unit run) has already been completed, typically by employing analyses at differing levels and of various types, on the generated data. Another example subcycle exists in the frequently exercised loop between building a model and executing it, as the user seeks to tune a model to specifications.

The research work that led to this dissertation, described in terms of BEAK, began with a desire to extend the executor (E–unit). Hooper and Reilly had developed a combined continuous and discrete (CCD) simulation system

(UAB–CIS's GGC processor) and an immediate goal was set to provide GGC with non–numeric (symbolic) simulation capabilities. An ultimate goal, designated for future research beyond this dissertation, was seen as providing a complete "combined continuous, discrete and symbolic" simulator (or more briefly, a "combined numeric–non–numeric simulator). The intention for the work reported in this dissertation was that it would provide many tools and a formal approach that would be important in aiding development of such a new combined simulator.

However, the emphasis on non–numeric computation, rule–based processing, and related processing meant that our work would be applicable to more than just the E–unit. An example, which we meet again shortly, concerns rule–based processing in providing guidance for users in selecting appropriate analysis routines. Another example we meet ahead is use of rule processing in a natural language input system; our group had already started work on this before this dissertation started.

These examples impact the A–unit and the B–unit, respectively. Others can be cited. The key point is that these considerations led to a conjecture about the BEAK which urges consideration of the entire BEAK in developing fundamental software. We amplify this conjecture, to include a role for our new formal methodology. This conjecture is meant to set a basis for the future work we propose; it is stated thus: In designing new software for simulation, the entire simulation environment should be taken into consideration. Formal development tools should apply to software development in all the individual elements of the environment.

That this conjecture is fruitful, we believe, is already and amply demonstrated in this dissertation. A major example is seen in the fact that software we designed with primary concern for the E–Unit very often has turned out useful for other units

as well. The table processing software can be cited in this regard. Our interest in this section is primarily on how future work can be based on work we and other workers at the UAB have done, of course, with the systems and software work described in this dissertation playing a key role. The BEAK categories provide a high level index for this purpose and also help bring a focus on work that spans categories.

The conjecture has a further impact in terms of forcing consideration of the advantages of a dual level formal approach (allowing modelers to reason at different levels in the programming hierarchy). This kind of contribution has already been made and it is demonstrated in this dissertation. Moreover, the conjecture will continue to propel us and our successors to coordinate practical software development with formal description of the software. This is a parallel development scheme: as the software tools increase so also does the formal description knowledge base. We have illustrated this kind of work in this thesis in order to make clear that such a parallelism is indeed possible. It seems reasonable now to recommend continuation of the methodology in future studies.

The conjecture additionally has promoted our taking a step toward defining and automating a part of the reasoning facility. Considerations of code automation and parallel development of tools and theory (as expressed in the last paragraph) are related items, as we have presented the matter in this dissertation.

As a result of the extra effort to investigate these issues we believe that the formal elements of our methodology are very close to being candidates for housing within an advanced simulation environment. The attack we have already mounted has resulted in ideas and feasibility demonstrations on components of simulation environments at the cutting edge in simulation. Let us now discuss at a more

detailed level some specific recommendations and suggestions for each of the units of the BEAK. A natural place to start is E–Unit activity.

14.6.1 E–Unit

We find that models under the labels symbolic, non–numeric, or AI, not infrequently turn out exclusively to be rule–based systems. The entire model may be consumed in a single rule–based system, with few other elements of modeling and simulation present. Occasionally, we see the *embedding* of expert system components into conventional numeric simulations, but less frequently do we see expert systems embedded within the more or less standard general purpose simulators. This circumstance may be due in part to the potential complexity of such systems, but, perhaps it is due more to the often unaccommodating posture these standard systems take to "foreign" elements.

We may quote A. Martin Wildberger at this point (though his comment covers more than just the case of complex model execution code):

> Current combinations [of AI and Simulation] seem awkward, and there is a real need for standardized ways to interface AI and Simulation techniques. [Wildberger, 1990].

Our hope for success in our foray into this territory where hitherto others have been moderately successful at best is predicated on several of our assumptions, e.g., that 1) having both numerical and non–numeric (AI) systems features coded in the same and consistent framework simplifies the combination; 2) having code publicly available for examination fosters adaptations for all users and systems developers; and 3) producing an *extensible* programming system is the appropriate approach to diversity of application and predicted future growth. Consistent with the allowance for growth is that the theory component of our systems is designed to apply to extensible systems.

E–Unit features should be developed from simulation *experience* as a guiding factor. An example is seen in our work where models adjoining numeric and non–numeric processing in psychobiology modeling [Reilly, Freese & Rowe, 1984; Reilly & Gfeller, 1976]) were a stimulus to adopting some features we have developed, and, in other cases, have legislated against selection. Among selected items are some tools facilitating string and list processing. Generalizing from these specifics to the excellent "models" for string and list processing facilities, workers may be able to draw on deep roots in computer theory, e.g., in Post Production Systems, in Markov Algorithms, in lambda calculus and in standard two–value logic theory. These banks of knowledge may provide help in further development of the theoretical side of our methodology. Some of the capabilities represented by theories have appeared in programming language realizations bearing familiar names of OPS5, Snobol, Lisp and Prolog. Additional guidance should derive from programming practice with these, as we have already done with various elements reported in this dissertation.

Our suggested approach follows our past modus operandi: to proceed eclectically, seeking out "good" features, prototyping, testing them, and employing the formal methodology as soon as ideas begin to suggest a modicum of persistence. We have not nor do we recommend merging all or even a substantial number of features from various sources, since incompatibilities and redundancies would then exist. In software development excursions in behalf of this dissertation, some of which are still in the "informal" (or "preformal") stage, we incorporated collections of related features into what we call "kits." The key software engineering concept we explored, then, was that, by having a single base language to which we can add (and subtract) features under ultimate guidance of formal theory, we solve some of

the potential problems of integrating programming and processing styles. We believe this approach should work well in others' (future) work.

14.6.2 A–Unit

Wildberger's comment about awkwardness of current combinations of numeric and non–numeric processing is not restricted to E–units. A case involving A–units can be understood from the perspective of rule–based operations in the work of Mellichamp and Park [Mellichamp, 1989]. These researchers developed rule–based systems with conventional expert system shells; the resulting expert systems offer guidance in analyzing results obtained from simulations. Their work is of great interest within our group at the time of this writing, and accordingly is under scrutiny.

One principal concern is how analysis routines used in A–unit – K–unit subcycles might be constructed so that they are equally useful for analysis operations occurring in E–unit model runs. An example might be a statistic such as Chi–Square, known to be useful in post–run operations on frequency distributions, but also useful in stopping rules within model runs. Our software approach promotes migration of code, here from A–unit to E–unit, since all of the non–numeric and rule–processing code, as well as that of the numeric code in GGC, is written within a single language. Since we have already developed code and schematics for creating rule–based systems as well as processing them, we have set the stage for this kind of migration issue to be handled without major awkwardness. Again, we recommend attention to this issue for future study potential.

Another element of Mellichamp and Park's work is that, when the system is being employed by a user, rules *previously acquired* are center stage. This is not inappropriate in their case, since much guidance can be done before the fact, i.e.

before model runs even begin. However, it is not always the case that rules can be ascertained beforehand. Hence, it is instructive to contrast their work with some discussed a little later, where rules are acquired *during* BEAK action. Accordingly, we recommend issues relating to graceful augmentation of rule systems in relationship to roles that these systems play in simulation.

Another different kind of A–unit activity can be mentioned in connection with our work on graphics within the ASP system. The area is that of visualization. The impact of visualization is expected to be enormous in simulation. Our work within ASP is only a beginning and much more will be needed. Nevertheless, we can recommend studying our software system in a context of a powerful visualization sub– or co–system. The primary unit of concern is the A–unit, but once again, migration to the E–unit, for much the same reasons as in the Mellichamp and Park case, must be considered in future study.

14.6.3 K–Unit

Other contributions based on *experience* involving persons who have used our system are found in knowledge acquisition work of Dey and Reilly [Dey & Reilly, 1986; Reilly & Dey, 1987]. These documents report on a framework and data structures for obtaining expert information from a variety of sources, including simulation models. Among models considered was one designed to simulate decision processes that the simulation environment itself might use, and another one in which neural nets are used to analyze the internal working of a complex CCD model [Reilly & Oliver, 1988].

These studies implicated the GGC system and related software, including some of our own. Accordingly, they appear as good starting points for launching future work. Some very sophisticated conceptualizations can be tackled from a framework which assumes that understanding the systematics and code that make a

model faithfully mimic a modeled system is really only part of the need in a simulation environment.

Accordingly, we posit a framework which, for a successful BEAK, envisions inquiry into such matters as: how existing results can be used for extrapolation and interpolation, how one model leads to pursuit of another, and last but hardly least, how results from several individual modeling efforts can be accumulated. Computer scientists will be called upon increasingly to do the "systems" work for environments in which these considerations are addressed. We offer our suggestions in the belief that they are of value to such individuals.

14.6.4 B–Unit

Several other studies performed on simulation and closely related systems have a potential bearing on the UAB efforts on simulation environments. Charles Autrey [1984], for example, dealt with fuzzy systems concepts and "linguistic variables" for E–unit alternatives to the numerical solution of a complex queueing system. These models were based on results emerging from E– and A–Units (i.e., direct results from simulation model runs and results that are uncovered upon massaging statistically the direct results). The kinds of models Autrey was interested in may be viewed as alternatives to the precise modeling, say, of GGC. As such, the main focus of the study is E–unit, but for the longer term the process of building models (B–unit) based on complex associations within the K–unit will emerge as a focus.

Preston B. Rowe approached a problem which constitutes a possible first step in extracting information from K–units for use in a B–unit. The focus was on directly extracting fragments of models from existing stored models. The long–term potential included use of established results of those models; this research involves possible merging with simulation approaches such as that

espoused in the book by Sauer and MacNair [1983] where the specific (detailed) nature of the systems being modeled is incorporated into the simulator itself. The interface between general queueing (and queueing analogs) and specific queueing systems for a limited range of problems seems a worthy future study target for our software and the software methodology.

Kevin Ramer [1986] worked on natural language input for simulation systems. In this system, with appropriate help from a dictionary, we can introduce a service system model in the terminology of the system being studied, e.g., a "gasoline station"; the code generated is for a general systems model in GPSS. Ramer's problem and the problem mentioned at the end of the preceding paragraph bear some similarity. The natural language input problem, however, does not include any reference to results or their nature (i.e., statistical detail). The central issue is translation from the terminology of an applications area into a fixed vocabulary general systems simulator. The similarity is in the target, since the specific systems being studied lend themselves to a controlled vocabulary rendition.

We can summarize this section by noting that the mimicking capabilities of Barrel accommodates mechanisms used by these writers: the Pascal higher-level language features used by Rowe, the special purpose language constructions from Lisp used by Autrey, and the Prolog capabilities used in part of Ramer's efforts. Some of the facilities they used have already been incorporated into Barrel code. It thus seems possible to foresee work refining and further developing these features, in behalf of the simulation environment of the future.

14.6.5 BEAK Unit Interactions

Though most of the action being described in the last paragraphs is directed toward specific BEAK units, sophisticated BEAK unit interactions are often implied. Since existing knowledge guides the builder in what he seeks, what he

specifies and designs must be expressed to the E–unit, and the final post–mortem statistical and other analyses are handled in the A–unit as it massages the data and delivers reduced data over to the K–unit, the appeal of the conjecture on the BEAK appears again. It is hoped that our work, which has impact at various points in the BEAK, together with our conjecture about the entire process of software development, can provide guidance in tackling what should be clearly seen now as a difficult and long–enduring problem area.

These studies, as well as earlier ones, represent an adequate launching pad for further research, and much can be achieved using our, or augmentations of our, current software schemes and software development methodology. The software we have treated in this dissertation can help remedy many of the difficulties we perceive in sophisticated interactive BEAK unit systems, because it is developed to be completely compatible with the GGC system, which incorporates a GPSS– and GASP–like facility in flexible and portable languages such as Fortran and C (C $^{++}$ being a new focus).

Our rule–based systems capabilities are embedded in the C language, which gives us some benefits in flexibility. Some additional flexibilities also emerge from our stance that rule processing be cast in a table processing general style, such as that laid out by authors such as Montalbano [1974], McDaniel [1978], and Metzner and Barnes [1977]. We can also include small gestures toward flexibility in file processing because of our connection to relational databases as a form of table processing, due to our work's having been coordinated at least to some degree with that of A. Salah, who sought links between decision tables and relational databases. Indeed, his work and ours together can be viewed as a joint effort in development of software relevant to a modern view of the BEAK (see the fifth section of this chapter).

14.7 DEPARTING WORDS

In these last two chapters, we have provided some of the obligatory fare for dissertation last chapters. We have summarized the entire work, first in general and then point–by–point. Recitation of detailed results and their associated prognoses allows us to plot some futures, how existing systems might benefit from our methods, but most importantly, how we have enabled parallel research to chart untried territory. These deliberations have marked the last pages of this dissertation and have provided a fitting end to our text.

REFERENCES

1. Abelson, H. "A Beginner's Guide to Logo." Byte, 7(8), August, 1982, pp. 88–112.

2. "Ada: Past, Present, Future: An Interview with Jean Ichbiah, the Principal Designer of Ada." Communications of the ACM, 27(10), October, 1984, pp. 990–997.

3. Autrey, J. C. "Application of Fuzzy Set Theory to Simulation and Control." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1984.

4. Barrett, J. H. "Studies in Non–Numerical Software Development." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1981a.

5. Barrett, J. H. "Use of Multiple Processes with STAGE2." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1981b.

6. Barrett, J. H. "Use of Overlays with STAGE2." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1981c.

7. Barrett, J. H. "Use of Paging Techniques with STAGE2." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1981d.

8. Barrett, J. H. "Stage2 Extensions and Modifications." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1982.

9. Barrett, J. H. "Using Barrel/ASP's Decision Table Entry, Translation and Presentation System to Construct Tables for a C Decision Table Processor." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1983a.

10. Barrett, J. H. "Using GIGI Graphics With Barrel/ASP's Decision Table Entry, Translation, and Presentation System." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1983b.

11. Barrett, J. H. and K. D. Reilly. "Non-Numerical Software Development Studies." Presented at the 19th Annual Southeast Regional ACM Conference, Atlanta, GA., April, 1981.

12. Barrett, J. H. and K. D. Reilly. "Realization of a Translator for Janus." Proceedings of the 20th Annual Southeast Regional ACM Conference, Knoxville, TN., April, 1982, pp. 223–225.

13. Barrett, J. H. and K. D. Reilly. "The Making of ASP: A Language Development Facility." Journal of the Alabama Academy of Science, 54(3), July, 1983, p. 192. (Abstract). Also presented at the 60th Annual Meeting of the Alabama Academy of Science, Tuscaloosa, AL., April, 1983.

14. Barrett, J. H. and K. D. Reilly. "Language Feature Analysis by ASP Implementations." Journal of the Alabama Academy of Science, 55(3), July, 1984, p. 229. (Abstract). Also presented at the 61st Annual Meeting of the Alabama Academy of Science, Mobile, AL., March, 1984.

15. Barrett, J. H. and K. D. Reilly. "Toward a Merger of Two UAB–Developed Programming Systems: GGC and Barrel–ELSDF." 25th Annual Conference of the Southeast Region of the Association for Computing Machinery, Birmingham, AL., April 1–3, 1987.

16. Barrett, J. H. and K. D. Reilly. "The Sixth Generation Neural Network Computing Meets the Fifth Generation Logic Programming." The Seventh Annual Southeastern Neuroscience Symposium, May 21, 1988 (poster session).

17. Brown, P. J. Macro Processors and Techniques for Portable Software. London: John Wiley and Sons, 1974.

18. Brown, P. J. ed. Software Portability. Cambridge: Cambridge University Press, 1977.

19. Brown, P. J. Writing Interactive Compilers and Interpreters. London: John Wiley and Sons, 1979.

20. Bruynooghe, M. Prolog in C for Unix Version 7. Louven, Belgium: Katholieke Universiteit, 1980.

21. Burge, W. Recursive Programming Techniques. Reading, Mass: Addison–Willey, 1975.

22. Burstall, R. M. "Formal Description of Program Structures and Semantics in First Order Logic." In Machine Intelligence, 5, E.U.P, 1969, pp. 79–98.

23. Cleaveland, J. C. and R. C. Uzgalis. Grammars for Programming Languages. New York: Elsevier North–Holland, 1977.

24. Clocksin, W. F. and C. S. Mellish. Programming in Prolog. New York: Springer-Verlag, 1981.

25. Coleman, S. S., P. C. Poole and W. M. Waite. "The Mobile Programming System, Janus." Software - Practice and Experience, 4(1), January - March, 1974, pp. 5-23.

26. Colmerauer, A. "Metamorphosis Grammars." In Natural Language Communication With Computers, New York: Springer-Verlag, 1978.

27. Davis, R. E. "Runnable Specification as a Design Tool." In Logic Programming, New York: Academic Press, 1982.

28. Dellert, G. T. "Improvements to TAB40 Decision Table Processor." MITRE Technical Report, MTR-6264, October, 1972.

29. Dey, P. and K. D. Reilly. "Integrating Knowledge Acquisition Methods." Proceedings of the 1986 IEEE International Conference on Systems, Man, and Cybernetics, Atlanta, GA., October, 1986.

30. Dijkstra, E. W. A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall, 1976.

31. Dilworth, R. "Programming with Higher-Order Functions in Lispkit Lisp." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1983.

32. Eades, C., K. Reilly, J. Barrett, and C. Minderhout. "The Barrel Concept: A Study in Language System Development." Proceedings of the 20th Southeast Regional ACM Conference, April, 1982, pp. 168-171.

33. Elrod, M. Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1981.

34. Gane, C. and T. Sarson. Structured Systems Analysis: Tools and Techniques. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

35. Gibson, J. "A Study in Software Portability: Implementation of the Stage2 Macro Processor on the VAX 11/750 in Pascal." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1982.

36. Gordon, M. J. C. The Denotational Description of Programming Languages: An Introduction. New York: Springer-Verlag, 1979.

37. Gratte, I. Starting With COMAL. Englewood Cliffs, N.J.: Prentice-Hall, 1984.

38. Griswold, R. E. "Linguistic Extension of Abstract Machine Modelling to Aid Software Development." Software – Practice and Experience, 10(1), January, 1980, pp. 1-9.

39. Griswold, R. E. "The Evaluation of Expressions in Icon." ACM Transactions on Programming Languages and Systems, 4(4), October, 1982, pp. 563–584.

40. Griswold, R. E. and M. T. Griswold. The Icon Programming Language. Englewood Cliffs, N.J.: Prentice-Hall, 1983.

41. Haddon, B. K. and W. M. Waite. "Experience with the Universal Intermediate Language Janus." Software – Practice and Experience, 8(5), September – October, 1978, pp. 601–616.

42. Haddon, B. K. and W. M. Waite. "The Universal Intermediate Language Janus (Draft Definition)." Technical Report, SEG–78–3, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, CO., 1978.

43. Harvey, B. "Why Logo?" Byte, 7(8), August, 1982, pp. 163–193.

44. Henderson, P. Functional Programming: Application and Implementation. Englewood Cliffs, N.J.: Prentice-Hall, 1980.

45. Hoare, C. A. R. "The Emperor's Old Clothes." Communications of the ACM, 24(2), February, 1981, pp. 75–83.

46. Hoare, C. A. R. and P. E. Lauer. "Consistent and Complimentory Formal Theories of Programming Languages." Acta Informatica, 3, 1974, pp. 135–153.

47. Hooper, J. W. and K. D. Reilly. "The 'GPSS–GASP Combined' (GGC) System." International Journal of Computer and Information Sciences, 12(2), 1983, pp. 111-136.

48. Hull, L. S., T. Takaoka, W. T. Jones, and B. R. Bryant. "Stack and Queue Programming Using SQ–Pascal." Technical Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1985.

49. Humby, E. Programs From Decision Tables. London: Mac Donald, 1973.

50. Johnson, S. C. Bell Laboratories, personal conversation, 1982.

51. Keller, J. F. and R. W. Roesch. "A Decision Logic Table Preprocessor." Masters Thesis, Naval Postgraduate School, U.S. Department of Commerce, Technical Information Service, AD A041154, Washington, D.C., 1977.

52. Kowalski, R. A. Logic for Problem Solving. New York: North Holland, 1979.

53. Kowalski, R. A. "AI and Software Engineering." Datamation, November, 1984a, pp. 92–102.

54. Kowalski, R. A. "Logic Programming in the Fifth Generation." Knowledge Engineering, Rev. 1, 1984b, pp. 26–38.

55. Lawler, R. W. "Designing Computer-Based Microworlds." Byte, 7(8), August, 1982, pp. 138–160.

56. Marcotty, M., H. F. Ledgard and G. V. Bachman. "A Sampler of Formal Definitions." Computing Surveys, 8(2), June, 1976, pp. 191–276.

57. McDaniel, H. An Introduction to Decision Logic Tables. New York: Petrocelli, 1978.

58. Metzner, J. R. and B. H. Barnes. Decision Table Languages and Systems. New York: Academic Press, 1977.

59. Mellichamp, J. M. and Y. H. Park. "A Statistical Expert System for Simulation Analysis." Simulation, 52(4), 1989, pp. 134–139.

60. Mills, H. D. "Program Design Without Arrays and Pointers." Technical Report, IBM Corporation, 1983.

61. Minderhout, C. and K. D. Reilly. "A Decision Table Entry, Translation, and Presentation System: An Applications Study in Barrel." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1982.

62. Minderhout, C., K. D. Reilly, J. H. Barrett, and J. Gibson. "Use of Barrel in Applications Studies." Proceedings of the 20th Annual Southeast Regional ACM Conference, April, 1982, pp. 172–175.

63. Montalbano, M. Decision Tables. Chicago: Science Research Associates, 1974.

64. Moss, C. D. S. "A Formal Definition of ASPLE Using Predicate Logic." Research Report DOC 80/18, Department of Computing, Imperial College, London, October, 1980.

65. Moss, C. D. S. "The Formal Description of Programming Languages using Predicate Logic." PhD Thesis, Department of Computing, Imperial College, London, 1981.

66. Moss, C. D. S. "How to define a language using PROLOG." Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming, August, 1982, pp. 67–73.

67. Neuhold, E. J. ed. Formal Description of Programming Concepts. Amsterdam: North-Holland Publishing Company, 1978.

68. Newey, M. C., P. C. Poole and W. M. Waite. "Abstract Machine Modelling to Produce Portable Software - A Review and Evaluation." Software - Practice and Experience, 2(2), April - June, 1972, pp. 107-136.

69. Orgass, R. J. and W. M. Waite. "A Base for a Mobile Programming System." Communications of the ACM, 12(9), September, 1969, p. 507.

70. Pagan, F. G. Formal Specification of Programming Languages: A Panoramic Primer. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

71. Papakonstantinou, G. "A Recursive Algorithm for the Optimal Conversion of Decision Tables." Angewandte Informatik, 22(9), September, 1980, pp. 350-354.

72. Pereira, F. ed. C-Prolog User's Manual - Version 1.4. SRI International, Menlo Park, CA., October, 1983.

73. Pereira, F. C. N. and D. H. Warren. "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks." Artificial Intelligence, 13, 1980, pp. 231-278.

74. Ramer, K. "Natual Language and Internal Representations in a Model Specification System." Master of Science Degree Project Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1986.

75. Reilly, K. D. and E. Gfeller. "Development and Use of Simulation Models in the Study of Abnormal Behavior." In: L. E. Gess, V. M. Heier and G. L. Berosik, Eds. Record of Proceedings: The Ninth Annual Simulation Symposium, 1976, pp. 61-82.

76. Reilly, K. D. and J. H. Barrett. "A Computerized Formal Methodology for Development and Modification of Numeric and Symbolic Components for Simulation Models and Environments." In: Jack Clema, Ed. Proceedings of the 1989 Summer Computer Simulation Conference, Austin, TX., July 24-26, 1989, pp. 550-555.

77. Reilly, K. D., J. H. Barrett, and H. Lilly. "Combined Continuous, Discrete, and Symbolic Simulation in a Parallel Processing Machine Context." In: Jordan Q. B. Chou, Ed. Proceedings of the 1987 Summer Computer Simulation Conference, Montreal, Quebec, Canada, July 27-30, 1987, pp. 73-78.

78. Reilly, K. D., J. H. Barrett, and A. Salah. "A First Study on a Prolog-Mimicking Barrel Extended Entry Decision Table Presentation System." Working Paper,

Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1982.

79. Reilly, K. D. and P. Dey. "Simulation Environments and Automated Knowledge Acquisition." In: Jordan Q. B. Chou, Ed. Proceedings of the 1987 Summer Computer Simulation Conference, Montreal, Quebec, Canada, July 27–30, 1987, pp. 668–673.

80. Reilly K. D., M. Freese, and P. B. Rowe. "Simulation Models of Abnormal Behavior: A Program Approach." Behavioral Science - Journal of the Society for General Systems Research, 29(3), 1984, pp. 186–211.

81. Reilly, K. D., W. Jones, and P. Dey. "The Simulation Environment Concept: Artificial Intelligence Perspectives." In: W. M. Holmes, Ed. Artificial Intelligence and Simulation. San Diego: Society for Computer Simulation, 1985, pp. 29–34.

82. Reilly, K. D., W. Jones, H. E. Lyons, P. Payer, K. W. Ramer, and P. Dey. "The AISEME Project: Artificial Intelligence and Software Engineering in Modeling Environments" ISETT 1985 Conference, University of Michigan, Ann Arbor, MI., PRISE Ref. No. M0705, August, 1985. 19pp.

83. Reilly, K. D., W. Jones, J. H. Barrett, A. Salah, E. Strand, J. Autry, and P. B. Rowe. "Software Development Studies: A Simulation Environment Perspective." ISETT 1984 Conference, University of Michigan, Ann Arbor, MI., Ref. No. M0657, 16pp.

84. Reilly, K. D., M. McAnulty, F. Amthor, M. Wainer, P. Thurston, and M. Villa. "Neural Network Modeling and the Neuronal Robot." In: Jordan Q. B. Chou, Ed. Proceedings of the 1987 Summer Computer Simulation Conference, Montreal, Quebec, Canada, July 27–30, 1987, pp. 448–453.

85. Reilly, K. D. and J. Oliver. "A Neural Control Element in a Control Systems Application." Proceedings of the First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, Tullahoma, TN., June 2–3, 1988, pp. 507–513.

86. Reilly, K. D., K. W. Ramer, P. Dey, B. W. Suter, H. E. Lyons, and J. Byoun. "A Natural Language Component for a Modeling and Simulation Environment." In: J. Young, V. W. Ingalls and R. Hawkins, Simulation at the Frontiers of Science, San Diego: Society for Computer Simulation, 1986, pp. 83–88.

87. Reilly, K. D., A. Salah, and C. C. Yang. "A Logic Programming Perspective on Decision Table Theory and Practice." Data and Knowledge Engineering, 2, 1987, pp. 191–212.

88. Reinwald, L. T. and G. T. Dellert. "TAB40 for FORTRAN IV on the CDC 64/65/6600." Research Analysis Corp., McLean, VA., October, 1968.

89. Rustin, R. ed. Formal Semantics of Programming Languages. Englewood Cliffs, N.J.: Prentice-Hall, 1972.

90. Salah, A. "On Decision Table Representation in Logic." Technical Report, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1983.

91. Salah, A. "An Integration of Decision Tables and a Relational Database System into a Prolog Environment." PhD Thesis, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1986.

92. Salah, A. and K. D. Reilly. "A Reduction Methodology for A Differential Diagnosis Expert System." International Journal of Approximate Reasoning, 1, 1987, pp. 131–139.

93. Salah, A., K. D. Reilly, and C. C. Yang. "A Logic Programming Approach to Decision Table Definition and Implementation." Presented at the ACM Midsoutheast Conference, Gatlinburg, TN., Nov., 1984.

94. Salah, A., K. D. Reilly, and J. H. Barrett. "A First Study in the Use of Prolog for Presentation of Extended Entry Decision Tables." Working Paper, Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL., 1982.

95. Sauer, Charles H. and Edward A. MacNair. Simulation of Computer Communication Systems. Englewood Cliffs, N.J.: Prentice-Hall, 1983.

96. Schwartz, B. M. "LISP 1.5 Decision Tables Implemented for a Serial Computer and Proposed for Parallel Computers." SIGPLAN Notices, 6(8), September, 1971, pp. 93–103.

97. Soloway, E., J. Bonar, and K. Erlich. "Cognitive Strategies and Looping Constructs: An Empirical Study." Communications of the ACM, 20(11), November, 1983, pp. 853–860.

98. Strachey, C. and C. P. Wadsworth. "Continuations - A Mathematical Semantics for Handling Full Jumps." PRG-11, Programming Research Group, University of Oxford, 1974.

99. Waite, W. M. "A Language Independent Macro Processor." Communications of the ACM, 10(7), July, 1967, pp. 433–440.

100. Waite, W. M. "Building a Mobile Programming System." Computer Journal, 13(2), February, 1970a, p. 28.

101. Waite, W. M. "The Mobile Programming System: STAGE2." Communications of the ACM, 13(7), July, 1970b, pp. 415–421.

102. Waite, W. M. Implementing Software for Non–Numeric Applications. Englewood Cliffs, N.J.: Prentice–Hall, 1973.

103. Waite, W. M. "Janus Memory Mapping: The J1 Abstraction." Technical Report, SEG–76–1, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, CO., 1976.

104. Waite, W. M. "Janus Stack Mapping: the J2 Abstraction." Technical Report, SEG–78–1, Software Engineering Group, Department of Electrical Engineering, University of Colorado, Boulder, CO., 1978.

105. Walters, R. F., J. Bowie, and J. C. Wilcox. MUMPS Primer: An Introduction to the Interactive Programming System of the Future. College Park, MD.: MUMPS Users' Group, 1982.

106. Warren, D. H. D., L. M. Pereira, and F. Pereira. "Prolog – The Language and Its Implementation Compared With Lisp." Sigplan Notices, 12(18), 1977.

107. Weiss, S. and C. Kulikowski. A Practical Guide to Designing Expert Systems Totowa, N.J.: Rowman & Allanheld, 1984.

108. Wildberger, A. M. "Artificial Intelligence and Simulation." Simulation, 55(1), 1990.

109. Wixson, S. University of Alabama at Birmingham, personal conversation, 1986.

APPENDIX 2.1

BARREL/ASP BARREL–F LANGUAGE

Informal description of the statements available in the Barrel–F programming language. Keywords are lower case; non–terminals are upper case. A "general case" example is followed by a specific example.

1. setq – assigns a value to a variable
   example: (setq VARIABLE EXP)
   (setq var 'some text)
   (setq var a+3*b)
   (setq var anothervar)

2. zero – sets the value of the variable to zero
   example: (zero VARIABLE)
   (zero var)

3. bump – adds the value of the expression to the value of the variable and makes that the new value of the variable
   example: (bump VARIABLE EXP)
   (bump var 2*c)

4. incr – increment the value of the variable by one
   example: (incr VARIABLE)
   (incr var)

5. decr – decrement the value of the variable by one
   example: (decr VARIABLE)
   (decr var)

6. execute – execute the value of the variable as if it were a statement
   example: (execute VARIABLE)
   (execute var)

7. readch – get an input value from channel 4 (as defined by ASP) and make it the new value of the variable
   example: (readch '4' VARIABLE)
   (readch '4' var)

8. ty – output the value of the variable or the quoted value; if there is more than one argument (separated by commas) concatenate them before output; arithmetic expressions are not supported
   example: (ty EXP[,EXP...])
   (ty 'the value of a is, a)

9. cbk – determine if the value of the variable is a valid condition or action (i.e., suitable for inclusion in a codebook as defined for the Barrel/ASP Decision Table Entry, Translation, and Presentation System); if not valid, print an error message and set the value of the variable "err" to 1; if valid, set the value of the variable "fh" to the length of the stub and set the value of the variable "bh" to the length of the longest entry (fh and bh are only set if their values are less than the values just found, i.e., if the values are bigger than any other found so far in the program)
   example: (cbk VARIABLE)
   (cbk var)

10. scn – output the contents of an array; the array name is given by the first variable, the starting point (either the zero or first element) is given by the integer, and the ending point is given by the value of the second variable
   example: (scn VARIABLE VARIABLE LIT)
   (scn array length 1)

11. stinit – initialize or clear a stack giving it the specified number of elements
   example: (stinit STACK EXP)
   (stinit ast '20)

12. push – push a value onto a stack
   example: (push STACK EXP)
   (push ast 'a value)

13. pop – remove a value from a stack and make it the new value of a variable
   example: (pop STACK VARIABLE)
   (pop ast var)

14. stcopy – copy the values of one stack to another stack destroying the previous contents of the target stack
   example: (stcopy STACK1 to STACK2)
   (stcopy thisstack to thatstack)

15. qinit – initialize or clear a queue giving it the specified number of elements
    example: (qinit QUEUE EXP)
    (qinit aqu '20)

16. inqfront – insert a value onto the front of a queue
    example: (inqfront QUEUE EXP)
    (inqfront aqu a*b)

17. inqback – insert a value onto the back of a queue
    example: (inqback QUEUE EXP)
    (inqback aqu a*b)

18. remqfront – remove a value from the front of a queue and make it the new value of a variable
    example: (remqfront QUEUE VARIABLE)
    (remqfront aqu var)

19. remqback – remove a value from the back of a queue and make it the new value of a variable
    example: (remqback QUEUE VARIABLE)
    (remqback aqu var)

20. qty – output the contents of a queue from front to back
    example: (qty QUEUE)
    (qty aqu)

21. qcopy – copy the values of one queue to another queue destroying the previous contents of the target queue
    example: (qcopy QUEUE1 to QUEUE2)
    (qcopy thisq to thatq)

22. loop/leave/again – looping control structure; execute the statements until the relation is true (the statements are optional); the literal value specified in the loop statement must match that in the leave and again statements; loops can be nested
    example: (loopLIT)
    STATEMENTS
    (if (EXP BOOLOP EXP) then leaveLIT)
    STATEMENTS
    (againLIT)

    (loop1)

```
(readch '4' var)
(if var eq 'end' then leave1)
(setq a a+var)
(again1)
```

23. if/then/else – if the relation is true then execute the first statement or statements, otherwise execute the second statement or statements (the else portion is optional); either form of else can be used with either form of then

```
example:  (if (EXP BOOLOP EXP) then STATEMENT)
          (else STATEMENT)

example:  (if (EXP BOOLOP EXP) then do)
           STATEMENTS
          (else do)
          STATEMENTS
          (endif)

          (if (a eq b) then (setq c '2))
          (else do)
          (setq c '2)
          (setq flag '1)
          (endif)
```

24. goto – the next statement to be executed is the one following the label statement

```
example:  (goto LIT)        where the label statement is:  (LIT:)
          (goto there)
          (setq a "his is not done)
          (there:)
```

25. stop – stop execution of the program

```
example:  (stop)
```

26. size – function; returns the precision if the value of the argument is a number or returns the length if the value of the argument is a character string

```
example:  (size EXP)
          (setq a (size 'this string))
          (setq b (size a*b+c))
```

27. concat – function; returns the concatenation of the character string values of the two arguments

example: (concat EXP EXP)
(setq a (concat a 'this string))

28. top – function; returns the value currently on top of the stack without popping it
example:(top STACK)
(setq var (top st1))

29. stsize – function; returns the number of values currently on the stack
example:(stsize STACK)
(ty (stsize st1),' elements)

30. stempty – function; returns true if the stack has no values on it and returns false otherwise
example: (stempty STACK)
(if ((stempty st1) eq 'true) then (goto empty))

31. front – function; returns the value currently on the front of the queue without removing it
example: (front QUEUE)
(ty (front aqu))

32. back – function; returns the value currently on the back of the queue without removing it
example: (back QUEUE)
(ty (back aqu))

33. qsize – function; returns the number of values currently on the queue
example: (qsize QUEUE)
(setq a (qsize aqu)+3)

34. qempty – function; returns true if the queue has no values on it and returns false otherwise
example: (qempty QUEUE)
(if ((qempty aqu) eq 'true) then (goto empty))

35. ! – plink operator; allows specification of the index of an array variable (the index can evaluate to an integer or a character string)
example: (setq VARIABLE!EXP EXP)
(setq arr!'3' 'a value)
(ty 'the name is: ,arr!'name')

Definition of non–keywords used in examples above:

VARIABLE — a variable name which can consist of any sequence of characters which are balanced with respect to parentheses

EXP — can be an arithmetic expression
example: 2*(a+1)
or a function call
example: (size box)
or a variable name
example: netpay
or a quoted value (a literal)
example: 'the rain in Spain
NOTE: a quoted value is delimited by the closing parenthesis when used in a setq statement and by a comma or the closing parenthesis when used in a ty statement
NOTE: an arithmetic expression can involve the four arithmetic operations +, –, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)

BOOLOP — a boolean operator; can be any of:

eq	(string equality)
ne	(string inequality)
=	(equal)
< >	(not equal)
<	(less than)
>	(greater than)
< =	(less than or equal)
= <	(less than or equal)
> =	(greater than or equal)
= >	(greater than or equal)

NOTE: eq and ne assume their arguments are strings and the other relational operators assume their arguments are integers

STATEMENT — can be any single statement

STATEMENTS — can be any sequence of zero or more statements
NOTE: each statement must fit on one line (usually 80

	characters but implementation dependent) so all non–keywords have an implied limit to their size
STACK	the name of a stack; see VARIABLE
QUEUE	the name of a queue; see VARIABLE
LIT	a literal constant value

APPENDIX 2.2

FEATURES AND FOLLIES OF THE BARREL–F FORMAL DEFINITION

Listed here are several strong points about the definition along with explanations of why they are worth mentioning. Then several weak points about the definition, brought out by thorough examination and testing, are expounded upon.

Strong points about the Barrel–F definition:

1. use of continuations for goto statement

The development of continuations by Strachey and Wadsworth was an important advance in the descriptive techniques of semantics. It led to simpler and smoother descriptions of various constructs, some of which would be impossible to describe without continuations [Gordon, 1979; Strachey & Wadsworth, 1974]. We have adapted the method of "impure continuations" to relational semantics in order to describe the goto statement, modeled after the work of Moss [1981]. The work was made somewhat easier by the fact that goto's cannot branch into or out of loops.

2. error messages match those of ASP

Errors are handled by a separate "status" parameter in the state of the machine. When an error occurs in the execution of a program the error message that is generated is the same as that received in the ASP implementation of Barrel–F.

3. execute statement sends the value of the variable through the 3 phases of the definition (lexical, syntactic, and semantic)

Barrel–F provides an execute statement which allows the value of a variable to be executed as if it were a Barrel–F statement. The semantic definition of the execute

statement actually sends the value of the variable through the 3 phases of the definition as if it were a one statement program.

4. implementation details can be included

Syntactic constraints which are machine dependent can be specified in the definition such as a limit on the size of integers. Checks can also be performed in the semantics to assure such constraints are followed at run time.

Problems with the Barrel-F definition:

1. statements are not limited to one line (80 characters)

The definition does not complain about statements that extend beyond the current line. The ASP implementation, however, does not allow statements to span more than one line. Nor does it allow more than 80 characters in a single line.

2. very large strings of digits are converted to floating point numbers upon input

In ASP, numbers are treated as character strings until an arithmetic operation is performed on them. Thus, very big numbers are allowed. But the definition (because of the way Prolog treats numbers) converts very long strings of digits to floating point numbers upon input. A similar problem occurs with leading zeros in numbers. The Prolog processor strips leading zeros upon input whereas the ASP processor allows them to be part of the number until an arithmetic operation is performed.

APPENDIX 2.3

FORMAL DEFINITION OF THE BARREL–F PROGRAMMING LANGUAGE

We first present a Prolog interface to the formal definition which makes its execution very simple. The user simply enters "go(file)." where file is the name of a file which contains a Barrel–F program. Each of the three parts of the definition (listed below) are called in turn: lexeme – the lexical syntax, morpheme – the syntax, and sememe – the semantics.

```
                    /*****************************/
                   /*****    main program    *****/
                  /*****************************/

/*** define the top level of the Barrel-F definition ***/
/*** the major predicates are lexemes, morpheme, and sememe ***/
go(File) :- see(File), read_in(Text), seen,
            lexemes(Tokens, Text, []), !,
            write('Tokens = '), pp(Tokens, 50, 9), nl, nl,
            morpheme([Tree | Mem], Tokens, []), !,
            write('Tree = '), pp(Tree, 50, 7), nl, nl,
            prettylist(Mem),
            write('Mem before sememe = '), pp(Mem, 50, 20), nl, nl,
            write('Enter your input in list form and end it
                  with a period: '),
            read(Input), nl,
            uglylist(Mem, Mem1), !,
            sememe(Tree, state(Mem1, [], Tree, Input, Output, ok),
                         state(M1, Cont, Tree, I1, O1, Result)),
            prettylist(M1),
            write('Mem after sememe = '), pp(M1, 50, 19), nl, nl,
            write('Input = '), pp(Input, 50, 8), nl, nl,
            prettylist(Output),
            write('Output = '), pp(Output, 50, 9), nl, nl,
            write('Result = '), write(Result), nl.

/*** read each character from a file into a list of ***/
/*** characters ***/
read_in([W | Ws]) :- get0(W), not checkeof(W), read_in(Ws), !.
```

```
read_in([]).
checkeof(26).  /* check for end of file */

/*** get rid of uninstantiated variable at the tail ***/
/*** of a list ***/
prettylist([]).
prettylist([Head | Tail]) :- var(Tail), Tail = [].
prettylist([Head | Tail]) :- prettylist(Tail).

/*** put uninstantiated variable at tail of a list ***/
/*** (opposite of prettylist) ***/
uglylist(List, []) :- isnull(List).
uglylist(List, Newlist) :- not isnull(List), putvar(List, Newlist).
putvar([X], [X | _]).
putvar([X | Y], [X | Z]) :- putvar(Y, Z).

/* Useful for printing long lists (more than 80 */
/* characters). It inserts newlines after every CPL */
/* characters and indents each line Sc characters. It */
/* uses file @@@ */
pp(List, CPL, Sc) :- not exists('@@@'), tell('@@@'),
                     write(List), told, see('@@@'),
                     pp1(1, CPL, Sc), seen, system("rm @@@").
pp(List, CPL, Sc) :- exists('@@@'), write(List).
pp1(Count, CPL, Sc) :- pp2(Count, CPL, Fl), (Fl = e;
                       nl, tab(Sc), pp1(Count, CPL, Sc)).
pp2(Count, CPL, Fl) :- CPL < 2, pp2(Count, 3, Fl).
pp2(Count, CPL, Fl) :- Count < CPL, get0(CH), (CH = 26, Fl = e;
                       put(CH), NewC is Count+1,
                       pp2(NewC, CPL, Fl)).
pp2(Count, CPL, Fl) :- Count = CPL, Fl = n.

/*** consult the other files needed for the Barrel-F ***/
/*** definition ***/
:- [lexemes, morpheme, syncon, sememe].

                    /*******************************/
                   /*****   lexical syntax   *****/
                  /*******************************/

/*** produce a list of tokens from the list of characters ***/
lexemes(X) --> [CH], {isnewline(CH)}, lexemes(X).
lexemes(X) --> comment, lexemes(X).
lexemes([X|Y]) --> token(X), lexemes(Y).
lexemes([]) --> [].

/*** get rid of comments ***/
comment --> [CH], {iseol(CH)}, restofcomment.
restofcomment --> [CH], {not isnewline(CH)}, restofcomment.
restofcomment --> [CH], {isnewline(CH)}.
```

```
/*** a token is a possible identifier ***/
token(id(Word)) --> word(W), {list(Word, W)}.
/* or any other character */
token(Other) --> [CH], {list(Other, [CH])}.

/*** build a word from alphanumeric characters ***/
word([First|Rest]) --> char(First), word(Rest).
/* quit when we get to a non-alphanumeric character */
word([Last]), [Next] --> char(Last), notchar(Next).

char(CH) --> [CH], {ischar(CH)} .
notchar(CH) --> [CH], {not ischar(CH)}.

                  /*******************************/
                 /*****   syntax portion   *****/
                /*******************************/

/*** build a Tree of abstract syntax statements and an ***/
/*** Environment of variables and their values ***/
morpheme([Tree | Env]) --> stmtrain(Env, Tree),
                           [' '], ['('], stopstm(_), [')'].

/*** process the statements of the program ***/
stmtrain(Env, [Sem | Seml]) --> [' '], ['('],
                                statement(Env, Sem), [')'],
                                (stmtrain(Env, Seml);
                                {Seml = []}).

/*** process an individual statement ***/
statement(Env, Sem) --> setqstm(Env, Sem);
                        zerostm(Env, Sem);
                        bumpstm(Env, Sem);
                        incrstm(Env, Sem);
                        decrstm(Env, Sem);
                        executestm(Env, Sem);
                        transputstm(Env, Sem);
                        cbkstm(Env, Sem);
                        scnstm(Env, Sem);
                        stackstm(Env, Sem);
                        queuestm(Env, Sem);
                        loopstm(Env, Sem);
                        ifstm(Env, Sem);
                        gotostm(Sem);
                        labelstm(Sem);
                        stopstm(Sem).

/*** assignment statement ***/
setqstm(Env, setq(Tag, Exp)) --> [id(setq)], [' '],
                                 identifier(Env, Tag, ' '),
                                 [' '], exp(Env, Exp, ')').
```

```
/*** set variable to zero statement ***/
zerostm(Env, setq(Tag, val(0))) --> [id(zero)], [' '],
                                    identifier(Env, Tag, ')').

/*** bump a variable by the value of an arithmetic expression ***/
bumpstm(Env, setq(Tag, plus(deref(Tag), Exp))) --> [id(bump)],
        [' '], identifier(Env, Tag, ' '), [' '],
        exp(Env, Exp, ')').

/*** increment the value of a variable ***/
incrstm(Env, setq(Tag, plus(deref(Tag), val(1)))) --> [id(incr)],
        [' '], identifier(Env, Tag, ')').

/*** decrement the value of a variable ***/
decrstm(Env, setq(Tag, minus(deref(Tag), val(1)))) --> [id(decr)],
        [' '], identifier(Env, Tag, ')').

/*** execute the value of a variable as a statement ***/
executestm(Env, execute(Exp)) --> [id(execute)], [' '],
                                  identifier(Env, Exp, ')').

/*** input and output statements ***/
transputstm(Env, output(Exp)) --> [id(ty)], [' '],
                                  outexp(Env, Exp), {!}.
transputstm(Env, input(Exp)) --> [id(readch)], [' '],
                                 {genquote(Quote)}, [Quote],
                                 [id(4)], [Quote], [' '],
                                 identifier(Env, Exp, ')').

/*** check for valid condition or action in the ***/
/*** codebook of a decision table ***/
cbkstm(Env, cbk(Tag)) --> [id(cbk)], [' '],
                          identifier(Env, Tag, ')'),
                          {declare(err, undef, Env)}.

/*** output the contents of an array ***/
scnstm(Env, scn(id(Tag), id(End), val(Beg))) --> [id(scn)], [' '],
       vn(Tag, ' '), [' '], vn(End, ' '), [' '], vn(Beg, ')').

/*** stack manipulation statements ***/
/* stinit statement (initialize stack) */
stackstm(Env, stinit(Stack, Exp)) --> [id(stinit)], [' '],
                                      stidentifier(Env, Stack, ' '),
                                      [' '], exp(Env, Exp, ')').
/* push statement */
stackstm(Env, push(Stack, Exp)) --> [id(push)], [' '],
                                    stidentifier(Env, Stack, ' '),
                                    [' '], exp(Env, Exp, ')').
/* pop statement */
stackstm(Env, pop(Stack, Tag)) --> [id(pop)], [' '],
```

```
                                    stidentifier(Env, Stack, ' '),
                                    [' '], identifier(Env, Tag, ')').
/* stack copy statement */
stackstm(Env, stcopy(Stack1, Stack2)) --> [id(stcopy)], [' '],
                                          stidentifier(Env, Stack1, ' '),
                                          [' '], [id(to)], [' '],
                                          stidentifier(Env, Stack2, ')').

/*** queue manipulation statements ***/
/* qinit statement (initialize queue) */
queuestm(Env, qinit(Queue, Exp)) --> [id(qinit)], [' '],
                                     qidentifier(Env, Queue, ' '),
                                     [' '], exp(Env, Exp, ')').
/* insert value in front of queue */
queuestm(Env, inqfront(Queue, Exp)) --> [id(inqfront)], [' '],
                                        qidentifier(Env, Queue, ' '),
                                        [' '], exp(Env, Exp, ')').
/* insert value in back of queue */
queuestm(Env, inqback(Queue, Exp)) --> [id(inqback)], [' '],
                                       qidentifier(Env, Queue, ' '),
                                       [' '], exp(Env, Exp, ')').
/* remove value from front of queue */
queuestm(Env, remqfront(Queue, Tag)) --> [id(remqfront)], [' '],
                                         qidentifier(Env, Queue, ' '),
                                         [' '],
                                         identifier(Env, Tag, ')').
/* remove value from back of queue */
queuestm(Env, remqback(Queue, Tag)) --> [id(remqback)], [' '],
                                        qidentifier(Env, Queue, ' '),
                                        [' '],
                                        identifier(Env, Tag, ')').
/* output the contents of a queue */
queuestm(Env, qty(Queue)) --> [id(qty)], [' '],
                              qidentifier(Env, Queue, ')').
/* copy from one queue to another */
queuestm(Env, qcopy(Queue1, Queue2)) --> [id(qcopy)], [' '],
                                         qidentifier(Env, Queue1, ' '),
                                         [' '], [id(to)], [' '],
                                         qidentifier(Env, Queue2, ')').

/*** looping statement (loop/if/then/leave/again) ***/
loopstm(Env, loop(S1, Exp, S2)) --> [id(Loop)],
       /* check for proper nesting */
       {checknest(Env, Loop, LoopEnv, Symb)},
       [')'], (stmtrain(LoopEnv, S1); {S1 = []}),
       [' '], ['('], [id(if)], [' '], relatexp(LoopEnv, Exp),
       [' '], [id(then)], [' '], [id(Leave)], [')'],
       /* check for proper nesting */
       {index(Leave, e, 1, 1, Ave),
```

```
         index(Ave, e, av, 2, Symb)},
        (stmtrain(LoopEnv, S2); {S2 = []}),
        [' '], ['('], [id(Again)],
        /* check for proper nesting */
        {index(Again, n, agai, 4, Symb)},
        /* leave a level of nesting */
        {leavenest(LoopEnv, Env)}.

/*** if/then/else statements ***/
ifstm(Env, if(Exp, S1, S2)) --> [id(if)], [' '],
                                relatexp(Env, Exp),
                                [' '], [id(then)], [' '],
                                thenpart(Env, S1),
                                elsepart(Env, S2).

/*** goto a different part of the program ***/
gotostm(goto(Label)) --> [id(goto)], [' '], [id(Label)].

/*** label (object of goto statement) ***/
labelstm(lab(Label)) --> [id(Label)], [:].

/*** stop execution of the program ***/
stopstm(stop) --> [id(stop)].

/*** get a variable name ***/
/* get an array name */
identifier(Env, id(array(Tag, Exp)), Endch) --> vn(Tag, '!'),
          ['!'], {not isnull(Tag)}, exp(Env, Exp, Endch),
          {declare(array(Tag), val([]), Env)}.
/* get a non-array name */
identifier(Env, id(Tag), Endch) --> vn(Tag, Endch),
                                    {declare(Tag, undef, Env)}.
/* get a stack name */
stidentifier(Env, id(stack(Tag)), Endch) --> vn(Tag, Endch),
            {declare(stack(Tag), val(undef, []), Env)}.
/* get a queue name */
qidentifier(Env, id(queue(Tag)), Endch) --> vn(Tag, Endch),
           {declare(queue(Tag), val(undef, []), Env)}.

/*** general expression handler (quoted strings and ***/
/*** arithmetic expressions) ***/
exp(Env, Exp, Endch) --> (quote(Exp, noout);
                          arithexp(Env, Exp, Endch)),
                         ({notequal(Exp, expr(error))};
                          {equal(Exp, expr(error))},
                           vn(_, Endch)).

/*** process quoted values found in assignment ***/
/*** statement expressions, array indices, and output ***/
/*** statement expressions ***/
```

```
quote(val(Val), Type) --> {genquote(Quote)}, [Quote],
                          qv(QL, 0, Type),
                          {list(Val, QL), !}.

/*** get quoted strings (generate list of ascii codes) ***/
qv([40 | R], PC, Type) --> ['('], {NPC is PC + 1},
                           qv(R, NPC, Type).
qv([], 0, Type), [Delim] --> delimit(Type, Delim).
qv([41 | R], PC, Type) --> [')'], {PC > 0, NPC is PC - 1},
                           qv(R, NPC, Type).
qv([], 0, Type) --> {genquote(Quote)}, [Quote].
qv(List, PC, Type) --> [id(Word)], {list(Word, L1)},
                       qv(L2, PC, Type),
                       {append(L1, L2, List)}.
qv(List, PC, Type) --> [Any], {list(Any, L1)},
                       qv(L2, PC, Type),
                       {append(L1, L2, List)}.

/*** have we reached the delimiter for the quoted string? ***/
/* output statements are delimited by closing parens and commas */
delimit(out, ')') --> [')'].
delimit(out, ',') --> [','].
/* non-output statement values are delimited by closing parens */
delimit(noout, ')') --> [')'].

/*** expression handler for arithmetic expressions ***/
arithexp(Env, Exp, Endch) --> factor(Env, Lh, Endch),
                              restexp(Env, Lh, Exp1, Endch),
                              /* unquoted numbers are treated */
                              /* identifiers */
                              ({equal(Exp1, val(Val)),
                                number(Val),
                                Exp = deref(id(Val))};
                               {Exp = Exp1});
                              {Exp = expr(error)}.
restexp(Env, Lh, Exp, Endch) --> [CH], {isaddsub(CH)},
                                 factor(Env, Rh, Endch),
                                 {op(CH, Lh, Rh, Subexp)},
                                 restexp(Env, Subexp, Exp, Endch).
restexp(Env, Lh, Lh, Endch) --> [].
factor(Env, Exp, Endch) --> primary(Env, Lh, Endch),
                            restfactor(Env, Lh, Exp, Endch).
restfactor(Env, Lh, Exp, Endch) --> [CH], {ismuldiv(CH)},
                                    primary(Env, Rh, Endch),
                                    {op(CH, Lh, Rh, Subexp)},
                                    restfactor(Env, Subexp, Exp, Endch).
restfactor(Env, Lh, Lh, Endch) --> [].
primary(Env, Exp, Endch) --> func(Env, Exp);
                             number(Exp);
```

```
                              expid(Env, Exp, Endch);
                              ['('], arithexp(Env, Exp, ')'), [')'].

/*** process functions ***/
/* handles the size function call */
func(Env, size(Exp)) --> ['('], [id(size)], [' '],
                         exp(Env, Exp, ')'), [')'].
/* handles the concat function call */
func(Env, concat(Exp1, Exp2)) --> ['('], [id(concat)], [' '],
                                  exp(Env, Exp1, ' '), [' '],
                                  exp(Env, Exp2, ')'), [')'].
/* handles the top of stack function call */
func(Env, top(Tag)) --> ['('], [id(top)], [' '],
                        stidentifier(Env, Tag, ')'), [')'].
/* handles the stack size function call */
func(Env, stsize(Tag)) --> ['('], [id(stsize)], [' '],
                           stidentifier(Env, Tag, ')'), [')'].
/* handles the stack empty function call */
func(Env, stempty(Tag)) --> ['('], [id(stempty)], [' '],
                            stidentifier(Env, Tag, ')'), [')'].
/* handles the front of queue function call */
func(Env, front(Tag)) --> ['('], [id(front)], [' '],
                          qidentifier(Env, Tag, ')'), [')'].
/* handles the back of queue function call */
func(Env, back(Tag)) --> ['('], [id(back)], [' '],
                         qidentifier(Env, Tag, ')'), [')'].
/* handles the size of queue function call */
func(Env, qsize(Tag)) --> ['('], [id(qsize)], [' '],
                          qidentifier(Env, Tag, ')'), [')'].
/* handles the queue empty function call */
func(Env, qempty(Tag)) --> ['('], [id(qempty)], [' '],
                           qidentifier(Env, Tag, ')'), [')'].

/*** do we have a number ***/
number(val(Val)) --> [id(Val)], {number(Val)}.
/* check for number preceded by unary minus */
number(val(Val)) --> ['-'], [id(Val1)], {number(Val1), Val is -Val1}.

/*** determine variable name for variables in arithmetic ***/
/*** expressions (they cannot contain arithmetic operators) ***/
/* process variables preceded by unary minus */
expid(Env, times(val(-1), deref(id(Tag))), Endch) --> ['-'],
      expvn(Tag, Endch), {not isnull(Tag), firstch(Tag, CH),
      not number(CH), notequal(CH, '(')}.
expid(Env, deref(id(array(Tag, Exp))), Endch) --> vn(Tag, '!'), ['!'],
                                                  {not isnull(Tag)},
                                                  exp(Env, Exp, Endch).
expid(Env, deref(id(Tag)), Endch) --> expvn(Tag, Endch),
                                      {not isnull(Tag),
```

```
                                        firstch(Tag, CH),
                                        not number(CH),
                                        notequal(CH, '(')}.
expvn('', Endch), [CH] --> [CH], {isaddsub(CH); ismuldiv(CH);
                          equal(CH, ')'); equal(CH, Endch)}.
expvn(Tag, Endch) --> [id(ID)], expvn(Tag1, Endch),
                        {concat(ID, Tag1, Tag)}.
expvn(Tag, Endch) --> [CH], {not isaddsub(CH), not ismuldiv(CH),
                       notequal(CH, ')'), notequal(CH, id(_)),
                       notequal(CH, Endch)},
                       expvn(Tag1, Endch), {concat(CH, Tag1, Tag)}.

/*** determine variable name for an identifier ***/
vn(Tag, Endch) --> ['('], vn(Tag1, ')'),
                   {concat('(', Tag1, Tag2),
                    concat(Tag2, ')', Tag3)}, [')'],
                    vn(Tag4, Endch),
                   {concat(Tag3, Tag4, Tag)}.
vn('', Endch), [Endch] --> [Endch].
vn(Tag, Endch) --> [id(ID)], vn(Tag1, Endch),
                   {concat(ID, Tag1, Tag)}.
vn(Tag, Endch) --> [CH], {notequal(CH, Endch),
                    notequal(CH, ')'), notequal(CH, '('),
                    notequal(CH, id(_)), notequal(CH, ' ')},
                    vn(Tag1, Endch), {concat(CH, Tag1, Tag)}.

/*** expression handler for ty (output) statement ***/
/*** creates a list of values or dereferenced ***/
/*** identifiers for output ***/
outexp(Env, []), [')'] --> [')'].
outexp(Env, List) --> [','], outexp(Env, List).
/* output quoted values */
outexp(Env, [F | R]) --> quote(F, out), outexp(Env, R).
/* output function values */
outexp(Env, [F | R]) --> func(Env, F), outexp(Env, R).
/* output variable values */
outexp(Env, [F | R]) --> (identifier(Env, Tag, ',');
                          identifier(Env, Tag, ')')),
                         {F = deref(Tag)}, outexp(Env, R).

/*** process relational expressions ***/
relatexp(Env, Exp) --> ['('], exp(Env, Exp1, ' '),
                       [' '], relop(Exp, Exp1, Exp2), [' '],
                       exp(Env, Exp2, ')'), [')'].
/* relational operators for string and numeric comparison */
relop(eq(Exp1, Exp2), Exp1, Exp2) --> [id(eq)].
relop(ne(Exp1, Exp2), Exp1, Exp2) --> [id(ne)].
relop(neq(Exp1, Exp2), Exp1, Exp2) --> ['<'], ['>'].
relop(le(Exp1, Exp2), Exp1, Exp2) --> ['<'], ['='].
```

```
relop(le(Exp1, Exp2), Exp1, Exp2) --> ['='], ['<'].
relop(ge(Exp1, Exp2), Exp1, Exp2) --> ['>'], ['='].
relop(ge(Exp1, Exp2), Exp1, Exp2) --> ['='], ['>'].
relop(equ(Exp1, Exp2), Exp1, Exp2) --> ['='].
relop(lt(Exp1, Exp2), Exp1, Exp2) --> ['<'].
relop(gt(Exp1, Exp2), Exp1, Exp2) --> ['>'].

/*** process the then part of the if/then/else statement ***/
/* thenpart can be either one statement or a series of statements */
thenpart(Env, S1) --> ['('], statement(Env, S1), [')'].
thenpart(Env, S1) --> [id(do)], [')'], stmtrain(Env, S1).

/*** process the else part of the if/then/else statement ***/
/* elsepart can be present or omitted */
elsepart(Env, S2) --> [')'], [' '], ['('], [id(else)],
                      [' '], whichelse(Env, S2).
elsepart(Env, S2) --> [' '], ['('], [id(else)],
                      [' '], whichelse(Env, S2).
elsepart(Env, []) --> [' '], ['('], [id(endif)].
elsepart(Env, []) --> [].
/* else can be one statement or a series of statements */
whichelse(Env, S2) --> ['('], statement(Env, S2), [')'].
whichelse(Env, S2) --> [id(do)], [')'], stmtrain(Env, S2),
                       [' '], ['('], [id(endif)].

              /*********************************/
             /*****   semantic portion   *****/
            /*********************************/

/*** process list of abstract syntax statements ***/
sememe([S1 | S2], state(M, Cont, T, I, O, R), St2) :-
       sememe(S1, state(M, [S2 | Cont], T, I, O, R), St1),
       continuation(St1, St2).
sememe([]) --> [].

/*** process the individual abstract syntax statements ***/
sememe(setq(id(Tag), Exp)) --> sememe(Exp, Val),
                               ({equal(Val, error)},
                                update(Tag, val(0));
                               update(Tag, Val)).
sememe(execute(Exp)) --> sememe(Exp, id(Tag)),
                         lookup(Tag, Val), execute(Val).
sememe(input(Exp)) --> sememe(Exp, id(Tag)),
                       (transput(in, Val), update(Tag, Val);
                        iocherror).
sememe(output(List)) --> outval(List, '', Val),
                         transput(out, Val).
sememe(cbk(id(Tag))) --> lookup(Tag, Val), cbk(Val).
sememe(scn(id(Array), id(Tag), Num)) -->
       (lookup(Tag, Val); {Val = val(0)}),
```

```
        (lt(Val, val(2), val(true)),
        {End = val(1)}; {End = Val}),
        (eq(Num, val(0), val(true)), {Index = val(1)};
        {Index = val(0)}),
        scn(Array, Index, End).
sememe(stinit(id(Stack), Exp)) --> sememe(Exp, Max),
                                   update(Stack, val(Max, [])).
sememe(push(id(Stack), Exp)) --> sememe(Exp, Val),
                                 push(Stack, Val).
sememe(pop(id(Stack), id(Tag))) --> pop(Stack, Tag).
sememe(stcopy(id(Stack1), id(Stack2))) -->
       lookup(Stack1, val(Max1, Stvals1)),
       ({equal(Max1, undef)},
         stackerror(Stack1, 'not initialized, stcopy ignored');
        {notequal(Max1, undef)},
         lookup(Stack2, val(Max2, Stvals2)),
         ({equal(Max2, undef)},
           stackerror(Stack2, 'not initialized, stcopy ignored');
          {notequal(Max2, undef), length(Stvals1, 0, Len1)},
           (gt(val(Len1), Max2, val(true)),
             stackerror(Stack2, 'overflow occurred, stcopy ignored');
            le(val(Len1), Max2, val(true)),
             update(Stack2, val(Max2, Stvals1))))).
sememe(qinit(id(Queue), Exp)) --> sememe(Exp, Max),
                                  update(Queue, val(Max, [])).
sememe(inqfront(id(Queue), Exp)) --> sememe(Exp, Val),
                                     inq(Queue, Val, front).
sememe(inqback(id(Queue), Exp)) --> sememe(Exp, Val),
                                    inq(Queue, Val, back).
sememe(remqfront(id(Queue), id(Tag))) --> remq(Queue, Tag, front).
sememe(remqback(id(Queue), id(Tag))) --> remq(Queue, Tag, back).
sememe(qty(id(Queue))) --> lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef)},
         qerror(Queue, 'not initialized, qty ignored');
        qty(Qvals)).
sememe(qcopy(id(Queue1), id(Queue2))) -->
       lookup(Queue1, val(Max1, Qvals1)),
       ({equal(Max1, undef)},
         qerror(Queue1, 'not initialized, qcopy ignored');
        {notequal(Max1, undef)},
         lookup(Queue2, val(Max2, Qvals2)),
         ({equal(Max2, undef)},
           qerror(Queue2, 'not initialized, qcopy ignored');
          {notequal(Max2, undef), length(Qvals1, 0, Len1)},
           (gt(val(Len1), Max2, val(true)),
             qerror(Queue2, 'overflow occurred, qcopy ignored');
            le(val(Len1), Max2, val(true)),
             update(Queue2, val(Max2, Qvals1))))).
```

```
sememe(loop(S1, Exp, S2)) -->
       getstate(tree, SaveT), getstate(result, SaveR),
       ({equal(SaveR, looping)};
        newstate(result, looping),
        newstate(tree, [S1, Exp, S2])),
       sememe(S1),
       (checkstop;      /* check for a stop statement */
        checkio;        /* check for ioch error */
        (getstate(result, jumpover), newstate(result, looping),
          {Flag = f};
         sememe(Exp, val(true)), {Flag = t};
         sememe(Exp, val(false)), {Flag = f},
         sememe(S2)),
        ({equal(Flag, t)}, newstate(result, SaveR),
          newstate(tree, SaveT);
         checkstop;
         checkio;
         (sememe(loop(S1, Exp, S2)),
           (checkstop;
            checkio;
            newstate(result, SaveR),
             newstate(tree, SaveT))))).
sememe(if(Exp, S1, S2)) --> sememe(Exp, val(true)),
                            sememe(S1);
                            sememe(Exp, val(false)),
                            sememe(S2).
sememe(goto(Label)) --> getstate(tree, Tree), getstate(result, R),
                        ({notequal(R, looping),
                         findcont(lab(Label), Tree, Newcont, Cond, _)},
                          ({equal(Cond, ok)},
                            newstate(continuation, [Newcont]),
                            newstate(result, ok);
                           {equal(Cond, ioch)},
                            newstate(continuation, []),
                            newstate(result, 'I/O error'),
                            sememe(Newcont),
                            (getstate(result, ok); iocherror);
                           {equal(Cond, null)}, iocherror);
                         {equal(R, looping),
                          findlevel(goto(Label), Tree, Level),
                          loopgoto(Label, Tree, Newcont, Cond, Level)},
                           ({equal(Cond, ok)},
                             getstate(continuation, [S | Rest]),
                             newstate(continuation, [Newcont | Rest]);
                            {equal(Cond, jump)},
                             getstate(continuation, [S | Rest]),
                             newstate(continuation, [Newcont | Rest]),
                             newstate(result, jumpover);
```

```
                              {equal(Cond, ioch)},
                               newstate(continuation, []),
                               sememe(Newcont), (checkio; iocherror))).
sememe(lab(Label)) --> [].
sememe(stop) --> newstate(continuation,[]), newstate(result, stopped).

/*** process the arithmetic expressions ***/
sememe(plus(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                  sememe(Exp2, Val2),
                                  {add(Val1, Val2, Val)};
                                  exprerror,
                                  {Val = error}.
sememe(minus(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                   sememe(Exp2, Val2),
                                   {subtract(Val1, Val2, Val)};
                                   exprerror,
                                   {Val = error}.
sememe(times(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                   sememe(Exp2, Val2),
                                   {mult(Val1, Val2, Val)};
                                   exprerror,
                                   {Val = error}.
sememe(division(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                      sememe(Exp2, Val2),
                                      {divide(Val1, Val2, Val)};
                                      exprerror,
                                      {Val = error}.

/*** process the relational expressions ***/
sememe(eq(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
                                eq(Val1, Val2, Val).
sememe(ne(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
                                ne(Val1, Val2, Val).
sememe(equ(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                 sememe(Exp2, Val2),
                                 equ(Val1, Val2, Val).
sememe(neq(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                 sememe(Exp2, Val2),
                                 neq(Val1, Val2, Val).
sememe(lt(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
                                lt(Val1, Val2, Val).
sememe(gt(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
                                gt(Val1, Val2, Val).
sememe(le(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
```

```
                                le(Val1, Val2, Val).
sememe(ge(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                sememe(Exp2, Val2),
                                ge(Val1, Val2, Val).

/*** process functions ***/
sememe(size(Exp), val(Val)) --> sememe(Exp, Val1),
                                {size(Val1, Val)}.
sememe(concat(Exp1, Exp2), val(Val)) -->
       sememe(Exp1, val(Val1)), sememe(Exp2, val(Val2)),
       {concat(Val1, Val2, Val)}.
/* process stack functions */
sememe(top(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = ''},
         stackerror(Stack, 'not initialized, returning null for top');
        {notequal(Max, undef), firstelem(Stvals, val(Val))}).
sememe(stsize(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = 0},
       stackerror(Stack, 'not initialized, returning zero for stsize');
        {notequal(Max, undef), length(Stvals, 0, Val)}).
sememe(stempty(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = true},
      stackerror(Stack, 'not initialized, returning true for stempty');
        {notequal(Max, undef)},
         ({isnull(Stvals), Val = true};
          {not isnull(Stvals), Val = false})).
/* process queue functions */
sememe(front(id(Queue)), val(Val)) -->
       lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef), Val = ''},
         qerror(Queue, 'not initialized, returning null for front');
        {notequal(Max, undef), firstelem(Qvals, val(Val))}).
sememe(back(id(Queue)), val(Val)) -->
       lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef), Val = ''},
         qerror(Queue, 'not initialized, returning null for back');
        {notequal(Max, undef), lastelem(Qvals, val(Val))}).
sememe(qsize(id(Queue)), val(Val)) -->
       lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef), Val = 0},
         qerror(Queue, 'not initialized, returning zero for qsize');
        {notequal(Max, undef), length(Qvals, 0, Val)}).
sememe(qempty(id(Queue)), val(Val)) -->
       lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef), Val = true},
```

```
        qerror(Queue, 'not initialized, returning true for qempty');
       {notequal(Max, undef)},
        ({isnull(Qvals), Val = true};
         {not isnull(Qvals), Val = false})).

/*** process all other expressions ***/
sememe(deref(Exp), Val) --> sememe(Exp, id(Tag)),
      (lookup(Tag, Val), {!}; {Val = val('')}).
sememe(id(Tag), id(Tag)) --> [].
sememe(val(Val), val(Val)) --> [].
sememe(expr(error), val(0)) --> exprerror.

/*** continue with next statement on continuation list ***/
continuation(state(M, [S2 | Cont], T, I, O, R), St2) :-
            sememe(S2, state(M, Cont, T, I, O, R), St2).
continuation(state(M, [], T, I, O, R), state(M, [], T1, I, O, R)).

/*** look up the value of a variable ***/
/* look up the value of an array variable */
lookup(array(Tag, Exp), Val) -->
       sememe(Exp, Index),
       getstate(memory, Mem),
       {lookup(array(Tag), Mem, val(List)),
       lookupa(Index, List, Val),
       notequal(Val, undef)}.
/* look up the value of an ordinary variable */
lookup(Tag, Val) --> getstate(memory, Mem),
                     {lookup(Tag, Mem, Val)}.
lookup(Tag, [loc(Tag, Val) | R], Val) :-
       notequal(Val, undef), not var(Val),
       (equal(Val, val(V)); equal(Val, val(V, _))),
       not var(V).
lookup(Tag, [loc(Tag1, V) | Rest], Val) :-
       notequal(Tag, Tag1),
       lookup(Tag, Rest, Val).
/* look up specific value of array variable for */
/* particular index */
lookupa(Index, [], undef).
lookupa(val(Index), [Index, Val | Rest], Val).
lookupa(Index, [Index1, Val1 | Rest], Val) :-
        lookupa(Index, Rest, Val).

/*** set a new value for a variable ***/
/* set a new value for an array variable */
update(array(Tag, Exp), Val) -->
      sememe(Exp, Index),
      getstate(memory, Mem1),
      {lookup(array(Tag), Mem1, val(List)),
      updatea(Index, Val, List, Newlist),
```

```
      update(array(Tag), val(Newlist), Mem1, Mem2)},
      newstate(memory, Mem2).
/* set a new value for an ordinary variable */
update(Tag, Val) --> getstate(memory, Mem1),
                     {update(Tag, Val, Mem1, Mem2)},
                     newstate(memory, Mem2).
update(Tag, Val, [], [loc(Tag, Val) | _]).
update(Tag, Val, [loc(Tag, V) | Env], [loc(Tag, Val) | Env]).
update(Tag, Val, [L | Env1], [L | Env2]) :-
       equal(L, loc(Tag1, V)),
       notequal(Tag, Tag1),
       (var(Env1),
         Env2 = [loc(Tag, Val) | _];
        update(Tag, Val, Env1, Env2)).
/* set specific value of array variable for a */
/* particular index */
updatea(val(Index), Newval, [], [Index, Newval]).
updatea(val(Index), Newval, [Index, Oldval | Rest],
                            [Index, Newval | Rest]).
updatea(Index, Newval, [Index1, Val1 | Rest],
                       [Index1, Val1 | Newrest]) :-
        notequal(Index, val(Index1)),
        updatea(Index, Newval, Rest, Newrest).

/*** execute a statement by sending it through all ***/
/*** three phases (lexical, syntax, and semantic) ***/
execute(Val, state(Mem, C, T, I, O, R),
             state(Mem1, C1, T, I1, O1, R1)) :-
        Val = val(Code), list(Code, Text),
        /* handle stop specially */
        (equal(Code, ' (stop)'), C1 = [], R1 = stopped,
          Mem1 = Mem, I1 = I, O1 = O;
         C1 = C, lexemes(Toks, Text, []),
         stmtrain(Mem, Tree, Toks, []), !,
         sememe(Tree, state(Mem, [], T, I, O, R),
                      state(Mem1, [], T, I1, O1, R1))).

/*** construct an output line for the ty statement ***/
outval([], Val, val(Val)) --> [].
outval([Exp | Rest], Val, NewVal) -->
       sememe(Exp, val(Val1)),
       {concat(Val, Val1, Val2)},
       outval(Rest, Val2, NewVal).

/*** produce a new input/output list ***/
transput(in, Val, state(Mem, C, T, In, O, R),
                  state(Mem, C, T, In1, O, R)) :-
                  io(Val, In, In1).
transput(out, Val, state(Mem, C, T, I, Out, R),
```

```
                    state(Mem, C, T, I, Out1, R)) :-
                    io(Val, Out, Out1).

/*** get an input value or add a new output value ***/
io(val(Val)) --> [val(Val)].

/*** process the cbk statement ***/
cbk(val(Val)) --> {index(Val, '[', Front, Len, Back)},
                  checkstub(Val, Front, Len, Err1),
                  ({equal(Err1, error)},
                   update(err, val(1));
                  {index(Back, ']', Entries, _, _)},
                  checkentries(Back, Entries, Err2),
                  ({equal(Err2, error)},
                   update(err, val(1));
                  lookup(fh, val(Vfh)),
                  ({Vfh >= Len};
                  update(fh, val(Len))))).

/*** check the stub of the codebook condition or action ***/
checkstub(Whole, Whole, Len, error) -->
          cbkerr('** error unbalanced or missing brackets').
checkstub(Whole, Stub, 0, error) -->
          cbkerr('** error no stub for condition or action').
checkstub(Whole, Stub, Len, error) --> {Len > 38},
          cbkerr('** error stub length > 38 chars.').
checkstub(Whole, Stub, Len, ok) --> {notequal(Whole, Stub),
          Len > 0, Len =< 38}.

/*** check the entries of codebook condition or action ***/
checkentries(Whole, Whole, error) -->
             cbkerr('** error unbalanced or missing brackets').
checkentries(Whole, Entries, Error) -->
             {index(Entries, ',', Entry, Len, Rest)},
             checklen(Len, Err1),
             ({equal(Err1, error), Error = error};
             lookup(bh, val(Val)),
             ({Val >= Len};
             update(bh, val(Len))),
             ({isnull(Rest), Error = ok};
             checkentries(Whole, Rest, Error))).

/*** check the length of an entry ***/
checklen(Len, error) --> {Len > 38},
         cbkerr('** error entry length > 38 chars.').
checklen(Len, ok) --> {Len =< 38}.

/*** process the scn statement ***/
scn(Tag, val(Index), val(End)) --> {Index > End}.
scn(Tag, val(Index), val(End)) --> {Index =< End},
```

```
    sememe(output([deref(id(array(Tag, val(Index))))])),
    {NewI is Index + 1}, scn(Tag, val(NewI), val(End)).

/*** push a value onto a stack ***/
push(Stack, Val) --> lookup(Stack, val(Max, Stvals)),
     ({equal(Max, undef)},
       stackerror(Stack, 'not initialized, push ignored');
      {notequal(Max, undef), length(Stvals, 0, Depth)},
       (equ(Max, val(Depth), val(true)),
         stackerror(Stack, 'stack overflow, push ignored');
        lt(val(Depth), Max, val(true)),
        update(Stack, val(Max, [Val | Stvals])))).

/*** pop a value onto a stack ***/
pop(Stack, Tag) --> lookup(Stack, val(Max, Stvals)),
    ({equal(Max, undef)},
      stackerror(Stack, 'not initialized, pop ignored');
     {notequal(Max, undef)},
     ({isnull(Stvals)},
       stackerror(Stack, 'pop on empty stack ignored');
      {not isnull(Stvals), Stvals = [Val | Rest]},
       update(Tag, Val), update(Stack, val(Max, Rest)))).

/*** insert a value onto the front or back of a queue ***/
inq(Queue, Val, ForB) --> lookup(Queue, val(Max, Qvals)),
    ({equal(Max, undef)},
      qerror(Queue, 'not initialized, queue insertion ignored');
     {notequal(Max, undef), length(Qvals, 0, Depth)},
      (equ(Max, val(Depth), val(true)),
        qerror(Queue, 'queue overflow, insertion ignored');
       lt(val(Depth), Max, val(true)),
        ({equal(ForB, front)},
          update(Queue, val(Max, [Val | Qvals]));
         {equal(ForB, back), append(Qvals, [Val], NewQvals)},
          update(Queue, val(Max, NewQvals))))).

/*** remove a value from the front or back of a queue ***/
remq(Queue, Tag, ForB) --> lookup(Queue, val(Max, Qvals)),
    ({equal(Max, undef)},
      qerror(Queue, 'not initialized, queue removal ignored');
     {notequal(Max, undef)},
      ({isnull(Qvals)},
        qerror(Queue, 'removal from empty queue ignored');
       {not isnull(Qvals)},
        ({equal(ForB, front), Qvals = [Val | Rest]},
          update(Queue, val(Max, Rest)), update(Tag, Val);
         {equal(ForB, back), lastelem(Qvals, Val)},
          update(Tag, Val), {rmlastelem(Qvals, NewQvals)},
          update(Queue, val(Max, NewQvals))))).
```

```
/*** output the contents of a queue ***/
qty([]) --> [].
qty([Val | Rest]) --> transput(out, Val), qty(Rest).

/*** a stop statement or a ioch error within a loop ***/
/*** statement must be handled specially ***/
checkstop --> getstate(result, stopped).
checkio --> getstate(result, 'I/O error').

/*** find a new continuation list for a goto statement ***/
/*** i.e. a new list of statements to do ***/
findcont(Stm, [], [], null, 1).
findcont(Stm, [Stm | Rest], Rest, ok, 1).
findcont(Stm, [loop(S1, Exp, S2) | Rest], Newcont, Cond, Level) :-
         findcont(Stm, S1, N1, C1, L1), equal(C1, ok),
         Newcont = N1, Cond = ioch, Level is L1 + 1;
         findcont(Stm, S2, N2, C2, L2), equal(C2, ok),
         Newcont = N2, Cond = ioch, Level is L2 + 1;
         findcont(Stm, Rest, Newcont, Cond, Level).
findcont(Stm, [if(Exp, S1, S2) | Rest], Newcont, Cond, Level) :-
         ifgoto(Stm, S1, C),
         (notequal(C, null), findcont(Stm, S1, N1, C1, L1),
          notequal(C1, null), append(N1, Rest, Newcont),
          Cond = C1, Level = L1);
         (findcont(Stm, S1, N2, C2, L2), notequal(C2, null),
          append(N2, S2, T), append(T, Rest, Newcont),
          Cond = C2, Level = L2);
         (findcont(Stm, S2, N3, C3, L3), notequal(C3, null),
          append(N3, Rest, Newcont), Cond = C3, Level = L3);
         findcont(Stm, Rest, Newcont, Cond, Level).
findcont(Stm, [X | Y], Z, Cond, Level) :-
         findcont(Stm, Y, Z, Cond, Level).

/*** find a new continuation list for a goto statement ***/
/*** in a loop (i.e. a new list of statements to do ***/
loopgoto(Label, [S1, Exp, S2], Newcont, Cond, Level) :-
         findcont(lab(Label), S1, N1, C1, L1),
         (notequal(C1, null), Newcont = N1,
           (equal(L1, Level), Cond = ok;
            notequal(L1, Level), Cond = ioch);
          findcont(lab(Label), S2, N2, C2, L2),
          (equal(C2, ok), Newcont = N2,
            (equal(L2, Level), Cond = jump;
             notequal(L2, Level), Cond = ioch);
           equal(C2, ioch), Newcont = N2,
            (equal(L2, Level), Cond = ok;
             notequal(L2, Level), Cond = ioch);
           Newcont = [], Cond = ioch)).
```

```
/*** determine if the goto is in the same if/then do group ***/
/*** as the label ***/
ifgoto(lab(Label), S1, Cond) :-
       findcont(goto(Label), S1, _, Cond, _).
ifgoto(Stm, S1, null).

/*** find the level of nested loop of a statement ***/
findlevel(Stm, [S1, Exp, S2], Level) :-
          findcont(Stm, S1, _, Cond, L),
          notequal(Cond, null), Level = L;
          findcont(Stm, S2, _, _, Level).

/*** get the current state of the machine ***/
getstate(memory, M, state(M, C, T, I, O, R),
                    state(M, C, T, I, O, R)).
getstate(continuation, C, state(M, C, T, I, O, R),
                          state(M, C, T, I, O, R)).
getstate(tree, T, state(M, C, T, I, O, R),
                  state(M, C, T, I, O, R)).
getstate(input, I, state(M, C, T, I, O, R),
                   state(M, C, T, I, O, R)).
getstate(output, O, state(M, C, T, I, O, R),
                    state(M, C, T, I, O, R)).
getstate(result, R, state(M, C, T, I, O, R),
                    state(M, C, T, I, O, R)).

/*** set a new state for the machine ***/
newstate(memory, Val, state(M, C, T, I, O, R),
                      state(Val, C, T, I, O, R)).
newstate(continuation, Val, state(M, C, T, I, O, R),
                            state(M, Val, T, I, O, R)).
newstate(tree, Val, state(M, C, T, I, O, R),
                    state(M, C, Val, I, O, R)).
newstate(input, Val, state(M, C, T, I, O, R),
                     state(M, C, T, Val, O, R)).
newstate(output, Val, state(M, C, T, I, O, R),
                      state(M, C, T, I, Val, R)).
newstate(result, Val, state(M, C, T, I, O, R),
                      state(M, C, T, I, O, Val)).

/*** generate an error message ***/
/* expression error */
exprerror --> transput(out, val('********* expr error')).
/* conversion error */
converror --> transput(out, val('********* conv error')).
/* input/output channel error (fatal) */
iocherror --> transput(out, val('********* ioch error')),
              newstate(continuation, []),
              newstate(result, 'I/O error').
```

```
/* output an error message for the codebook statement */
cbkerr(Errmess) --> transput(out, val(Errmess)).
/* output a stack error message */
stackerror(stack(Stack), Mess) -->
          sememe(output([val('** error on stack '),val(Stack),
                         val(': '),val(Mess)])).
/* output a queue error message */
qerror(queue(Queue), Mess) -->
       sememe(output([val('** error on queue '),val(Queue),
                      val(': '),val(Mess)])).

/*** perform the actual arithmetic operations ***/
add(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                     Z1 is X1 + Y1, inrange(Z1, Z).
subtract(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                          Z1 is X1 - Y1, inrange(Z1, Z).
mult(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                      Z1 is X1 * Y1, inrange(Z1, Z).
divide(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                        Z1 is X1 // Y1, inrange(Z1, Z).
/* make sure a number is in proper range (truncate if not) */
inrange(Int, Int) :- value(n1, Max), Int =< Max, Int >= -Max, !.
inrange(Num, Int) :- value(n2, Bits1), value(n3, Bits2),
                     Int is (Num<<(Bits2-Bits1))>>(Bits2-Bits1).

/*** perform the actual relational operations ***/
eq(X, Y, val(true)) --> {X == Y}.
eq(X, Y, val(false)) --> {not X == Y}.
ne(X, Y, val(true)) --> {not X == Y}.
ne(X, Y, val(false)) --> {X == Y}.
equ(X, Y, val(true)) --> checkrevals(X, Y, X1, Y1),
                         {X1 =:= Y1}.
equ(X, Y, val(false)) --> checkrevals(X, Y, X1, Y1),
                          {not X1 =:= Y1}.
neq(X, Y, val(true)) --> equ(X, Y, val(false)).
neq(X, Y, val(false)) --> equ(X, Y, val(true)).
lt(X, Y, val(true)) --> checkrevals(X, Y, X1, Y1), {X1 < Y1}.
lt(X, Y, val(false)) --> checkrevals(X, Y, X1, Y1), {X1 >= Y1}.
gt(X, Y, val(true)) --> checkrevals(X, Y, X1, Y1), {X1 > Y1}.
gt(X, Y, val(false)) --> checkrevals(X, Y, X1, Y1), {X1 =< Y1}.
le(X, Y, val(true)) --> gt(X, Y, val(false)).
le(X, Y, val(false)) --> gt(X, Y, val(true)).
ge(X, Y, val(true)) --> lt(X, Y, val(false)).
ge(X, Y, val(false)) --> lt(X, Y, val(true)).
/* check for valid arithmetic values for relational operators */
checkrevals(X, Y, X1, Y1) --> ({checkval(X, X1)};
                               {not checkval(X, X1), X1 = 0},
                               exprerror),
                              ({checkval(Y, Y1)};
```

```
                                {not checkval(Y, Y1), Y1 = 0},
                                exprerror).

/*** make sure a value is really a number ***/
checkval(val(''), 0).
checkval(val(Val), Val) :- number(Val).

/*** perform the size function ***/
size(val(Val1), Val) :- list(Val1, L), length(L, 0, Val).

                    /************************************/
                   /*****   syntactic constraints   *****/
                  /************************************/

/*** declare adds a declaration to the environment if ***/
/*** the variable is not already a member ***/
declare(Tag, Val, Env) :- member(loc(Tag, _), Env).
declare(Tag, Val, Env) :- not member(loc(Tag, _), Env),
                          addword(loc(Tag, Val), Env).

/*** check for proper nesting of loop statements ***/
checknest([loop(SL) | Rest], Loop,
          [loop([Symb | SL]) | Rest], Symb) :-
          not var(SL),
          index(Loop, p, loo, 3, Symb),  /* get loop symbol */
          not isnull(Symb), not member(Symb, SL).
checknest(Env, Loop, [loop([Symb]) | Env], Symb) :-
          var(Env), index(Loop, p, loo, 3, Symb),
          not isnull(Symb).
checknest(Env, Loop, [loop([Symb]) | Env], Symb) :-
          not var(Env), notequal(Env, [loop(SL) | Rest]),
          index(Loop, p, loo, 3, Symb), not isnull(Symb).

/*** remove level of nesting symbol from environment list ***/
/*** after a loop statement has been processed ***/
leavenest([loop([Symb]) | Env], Env).
leavenest([loop([Symb | Rest]) | Env], [loop(Rest) | Env]).

/*** member handles lists with uninstantiated tail ***/
member(X, [Y]) :- var(Y), !, fail.
member(X, [X | Y]).
member(X, [Y | Z]) :- notequal(X, Y), member(X, Z).

/*** add a word to the end of a list ***/
addword(Label, [X | Y]) :- var(X), var(Y), X = Label.
addword(Label, [X | Rest]) :- not var(X), addword(Label, Rest).

/*** append the 2nd list to the end of the 1st list ***/
/*** giving the 3rd list ***/
append([U | V], W, [U | X]) :- append(V, W, X).
append([], X, X).
```

```
/*** find the length of a list ***/
length([], N, N).
length([Head | Rest], Sofar, Total) :- More is Sofar + 1,
                                       length(Rest, More, Total).

/*** return the first element of a list ***/
firstelem([Head | Tail], Head).
firstelem([], '').

/*** find the last element of a list ***/
lastelem([Last], Last).
lastelem([X | Y], Last) :- lastelem(Y, Last).

/*** remove the last element of a list ***/
rmlastelem([Elem], []).
rmlastelem([F | R], [F | R1]) :- rmlastelem(R, R1).

/*** generate a single quote (apostrophe) character ***/
genquote(Quote) :- list(Quote, [39]).

/*** concatenate two atoms (Val1 and Val2) to form a ***/
/*** new atom (NewVal) ***/
concat('', Val2, Val2) :- !.
concat(Val1, '', Val1) :- !.
/* plus signs require special handling so we don't lose them */
concat('+', Val2, NewVal) :- integer(Val2),
                             NewVal =.. ['+', Val2], !.
concat(Val1, Val2, NewVal) :- isplus(Val1, Arg),
                              not isplus(Val2, _),
                              concat(Arg, Val2, Val3),
                              NewVal =.. ['+', Val3], !.
concat(Val1, Val2, NewVal) :- not isplus(Val1, _),
                              isplus(Val2, Arg),
                              concat(Val1, '+', Val3),
                              concat(Val3, Arg, NewVal), !.
concat(Val1, Val2, NewVal) :- isplus(Val1, Arg1),
                              isplus(Val2, Arg2),
                              concat(Arg1, '+', Val3),
                              concat(Val3, Arg2, Val4),
                              NewVal =.. ['+', Val4], !.
concat(Val1, Val2, NewVal) :- list(Val1, L1), list(Val2, L2),
                              append(L1, L2, L3),
                              list(NewVal, L3), !.

/*** extract the first character of an atom ***/
firstch(Atom, Ch) :- not isnull(Atom),
                     list(Atom, [First | Rest]),
                     list(Ch, [First]).

/*** find the single character (Char) in the string ***/
/*** (Str) and return the front of the string (Front), ***/
```

```
/*** the length of the front (Len), and the back of ***/
/*** the string (Back) (minus the character) ***/
index(Str, Char, Front, Len, Back) :- list(Str, StrL),
        list(Char, ChL), indexl(StrL, ChL, FrL, BkL, Len, 0),
        list(Front, FrL), list(Back, BkL).
indexl([CH | Rest], [CH], [], Rest, Count, Count).
indexl([], [CH], [], [], Count, Count).
indexl([First | Rest], [CH], [First | Front], Back, Total, Sofar) :-
       notequal(CH, First), NewC is Sofar + 1,
       indexl(Rest, [CH], Front, Back, Total, NewC).

/*** convert an atom to a list of ascii codes and vice ***/
/*** versa (same as built-in functor "name" but works ***/
/*** with null arguments) ***/
list('', []) :- !.
list(Atom, List) :- name(Atom, List).

isnull([]).              /* is arg. the null list */
isnull('').              /* is arg. the null string */
/* does Term contain an addition operation */
isplus(Term, Arg) :- Term =.. ['+', Arg].
/* is arg. the ascii code for a newline char. */
isnewline(10).
/* is arg. the ascii code for a end-of-line char. */
iseol(124).
/* is arg. one of the ascii codes for an alphanumeric char. */
ischar(CH) :- CH >= 65, CH =< 90.
ischar(CH) :- CH >= 97, CH =< 122.
ischar(CH) :- CH >= 48, CH =< 57.
isaddsub('+').           /* is arg. an addition sign */
isaddsub('-').           /* is arg. an subtraction sign */
ismuldiv('*').           /* is arg. an multiplication sign */
ismuldiv('/').           /* is arg. an division sign */

/*** convert an arithmetic operation to abstract notation ***/
op('+', Lh, Rh, plus(Lh, Rh)).
op('-', Lh, Rh, minus(Lh, Rh)).
op('*', Lh, Rh, times(Lh, Rh)).
op('/', Lh, Rh, division(Lh, Rh)).

/*** instantiate the args. to be the same thing (fail ***/
/*** if they are already instantiated to different things) ***/
notequal(X, Y) :- not(equal(X, Y)).
equal(X, Y) :- X = Y.

/*** various machine dependent values ***/
value(n1, 32767).     /* value of largest integer in ASP */
value(n2, 16).        /* bits per integer for ASP */
value(n3, 32).        /* bits per integer for this prolog */
```

APPENDIX 2.4

BARREL–F IMPLEMENTATION

Listed below are the macro definitions for the ASP implementation of the Barrel–F language.

```
 (setq #!'#' #)|
evalarg #30$
#10!#20#16 %arg#26 #21#26 #f3$
$
 (setq #!# #)|
evalarg #20$
%arg#26 #10!#21#16$
evalarg #30$
#21#26 #f3$
$
 (setq # #)|
evalarg #20$
%arg#26 #21#26 #f3$
$
 (zero #!#)|
evalarg #20$
%arg#96$
set #10!#91 to 0$
$
 (zero #)|
set #10 to 0$
$
 (bump #!'#' #)|
evalarg #30$
#10!#20#26 %arg#96$
number #91$
#21+#91#36$
set #20 to #34$
$
 (bump #!# #)|
evalarg #20$
%arg#96 #10!#91#26$
evalarg #30$
number #91$
#21+#91#36$
set #20 to #34$
```

```
$
 (bump # #)|
evalarg #20$
%arg#96$
number #91$
#11+#91#26$
set #10 to #24$
$
 (incr #!#)|
evalarg #20$
%arg#96$
#10!#91#96$
#91+1#86$
set #90 to #84$
$
 (incr #)|
#11+1#26$
set #10 to #24$
$
 (decr #!#)|
evalarg #20$
%arg#96$
#10!#91#96$
#91-1#86$
set #90 to #84$
$
 (decr #)|
#11-1#26$
set #10 to #24$
$
 (execute #!#)|
evalarg #20$
%arg#96 #10!#91#96$
#91$
$
 (execute #)|
#11$
$
 (readch '4' #!#)|
evalarg #20$
%arg#96 #10!#91#16$
2#fi4$
#f3$
$
 (readch '4' #)|
2#fi4$
#f3$
$
```

```
 (ty #)|
set %t to $
%arg#96$
%t#86$
(#10)#17,$
set %f to 0$
quotarg #10$
if %f eq '1' skip 5$
funarg #10$
if %f eq '1' skip 3$
arrayarg #10$
if %f eq '1' skip 1$
set %arg to #11$
set %t to #81#91$
#f8$
ty #81$
$
| checks action & conditions of codebook input data
$
 (cbk #!#)|
evalarg #20$
%arg#26 #10!#21#16$
x1#11$
$
 (cbk #)|
x1#11$
$
| screen routine: array name; no. elements; list 0 element
$
 (scn # # #)|
set %i to 0$
if #30 = 0 skip 2$
#10!0#96$
ty #91$
%i#96$
#21#f7$
setx %i to %i+1$
#10!#91#86$
ty #81$
#f8$
$
| stack statements - stack system variables prefixed by @
$
 (stinit # #)|        initialize/clear user stack
evalarg #20$
setv @#10 to %arg$    store maximum size of stack
set @#10sp to 0$      set stack pointer to zero
$
```

```
 (push # #)|          push the value #20 onto stack #10
if @#10 ne '' skip 2$
sterr #10<not initialized, push ignored>$
#f9$
evalarg #20$
if @#10sp < @#10 skip 2$
sterr #10<stack overflow, push ignored>$
#f9$
setx @#10sp to @#10sp+1$   increment stack pointer
@#10sp#86$
setv @#10#81 to %arg$      push value
$
 (pop # #!#)|        pop from stack (#10) and put in
evalarg #30$         array variable (#20!#30)
%arg#96$
upop #10 #20!#91$
$
 (pop # #)|          pop onto simple variable (#20)
upop #10 #20$
$
 (stcopy # to #)|    copy stack (#10) to stack (#20)
if @#10 ne '' skip 2$
sterr #10<not initialized, stcopy ignored>$
#f9$
if @#20 ne '' skip 2$
sterr #20<not initialized, stcopy ignored>$
#f9$
if @#10sp <= @#20 skip 2$
sterr #20<overflow occurred, stcopy ignored>$
#f9$
if @#10sp > 0 skip 2$
set @#20sp to 0$
#f9$
set %t to 0$               set temporary stack pointer
%t#96$
@#10sp#f7$
setx %t to %t+1$           increment stack pointer
setv @#20#91 to @#10#91$   copy a stack element
#f8$
setv @#20sp to @#10sp$     set new stack pointer
$
| queue statements - system queue variables preceded by &
$
 (qinit # #)|        initialize/clear queue
evalarg #20$               max size of queue
setv &#10 to %arg$         store max size of queue
setv &#10fr to %arg$       front of queue pointer
setx &#10ba to %arg+1$     back of queue pointer
```

```
$
 (inqfront # #)|     insert element (#20) in front of queue (#10)
inqfr #10 #20$
$
 (inqback # #)|      insert element (#20) in back of queue (#10)
inqba #10 #20$
$
 (remqfront # #!#)| remove element from front of queue (#10) and
evalarg #30$        put in array (#20!#30)
%arg#36$
remqfr #10 #20!#31$
$
 (remqfront # #)|   remqfront for simple variable (#20)
remqfr #10 #20$
$
 (remqback # #!#)|  remove element from back of queue (#10) and
evalarg #30$        put in array (#20!#30)
%arg#36$
remqba #10 #20!#31$
$
 (remqback # #)|    remqback for simple variable (#20)
remqba #10 #20$
$
 (qty #)|           output the contents of a queue (#10)
if &#10 ne '' skip 2$
qerr #10<not initialized, qty ignored>$
#f9$
if &#10ba-&#10fr-1 > 0 skip 1$   is queue empty?
#f9$
setx %t to &#10fr+1$        use %t as index into queue
%t#96$
&#10ba-&#10fr-1#f7$         loop "size of queue" times
&#10#91#86$
ty #81$
setx %t to %t+1$            increment index
#f8$                        end of loop
$
 (qcopy # to #)|    copy queue (#10) to queue (#20)
if &#10 ne '' skip 2$
qerr #10<not initialized, qcopy ignored>$
#f9$
if &#20 ne '' skip 2$
qerr #20<not initialized, qcopy ignored>$
#f9$
if &#10ba-&#10fr-1 <= &#20 skip 2$   will queue 1 fit in queue 2?
qerr #20<overflow occurred, qcopy ignored>$
#f9$
setv %t1 to &#10$           use %t1 as index into queue 1
```

```
setv %t2 to &#20$              use %t2 as index into queue 2
%t1#96 %t2#86$
&#10-&#10fr#f7$                loop through front of queue
setv &#20#81 to &#10#91$       copy an element
setx %t1 to %t1-1$             decrement index
setx %t2 to %t2-1$             decrement index
#f8$                           end of loop
setx %t1 to &#10+1$            point to back of queue 1
setx %t2 to &#20+1$            point to back of queue 2
&#10ba-&#10#f7$                loop through back of queue
setv &#20#81 to &#10#91$       copy an element
setx %t1 to %t1+1$             increment index
setx %t2 to %t2+1$             increment index
#f8$                           end of loop
setx &#20fr to &#20-(&#10-&#10fr)$   set new front of queue pointer
setx &#20ba to &#20+(&#10ba-&#10)$   set new back of queue pointer
$
|     control statements
$
 (loop#)|
push %ic '#10$
if %ic ne '2' skip 1$
#f9$
 (loop#10)#f12$
#10#96 %ic#86$
 (again#10)#16$
#81#f22$
#10#f12$
set %ic to 2$
repos  (loop#90)$     rewind chan 2 and pass over label
$
 (if (#) then leave#)|
if %ic eq '2' skip 1$
repos  $                  cause an ioch error
pop %t$
if %t eq '#20' skip 1$
repos  $
if relexp(#10) skip 2$
push %t$
#f9$
 (again#20)#16 %ic#86$    move past end of loop
#81#f20$
pop %ic$
if %ic eq '2' skip 3$     are we in a nested loop?
#16$
#f22r$                    no, rewind channel 2
text#81$
$
```

```
 (again#)|
if %ic eq '2' skip 1$
repos  $                    cause an ioch error
pop %t$
if %t eq '#10' skip 1$
repos  $
push %t$
repos  (loop#10)$
$
 (goto #)|
repos  (#10:)$
$
 (#:)|
$
 (if (#) then do)|
if relexp(#10) skip 3$
set %c to 0$
feoe$                       find else or endif
#f9$
eueoe$                      execute until else or endif
$
 (if (#) then (#))|
%ic#96 #91#96$
3#fi#90$
if relexp(#10) skip 4$
celd #30$                   condition is false, do else statement
if %tf eq 't' skip 1$
#30$
#f9$
 (#20)$                     condition is true, execute statement
cgoto  (#20)$               is statement a goto?
if %tf ne 't' skip 1$
#f9$                        yes, quit
celd #30$                   no, move past else portion
if %tf ne 't' skip 5$       is it an else do?
push %c$
set %c to 0$
feoe$                       yes, find the end
pop %c$
#f9$
cel #30$                    is else portion present?
if %tf eq 't' skip 1$       yes, ignore it
#30$                        no, execute next statement
$
 (else (#))|
 (#10)$
$
 (else do)|
```

```
$
 (endif)|
$
 (stop)|
#f0$
$
|
$      barrelf sub-definitions: definitions used by main
|                                      barrelf definitions
$
evalarg #| evaluate an argument for its type
set %f to 0$                  argument type flag
quotarg #10$                  is it a quoted value?
if %f eq '0' skip 1$
#f9$
aritharg #10$                 no, is it an arithmetic argument?
if %f eq '0' skip 1$
#f9$
funarg #10$                   no, is it a function?
if %f eq '0' skip 1$
#f9$
arrayarg #10$                 no, is it an array variable?
if %f eq '0' skip 1$
#f9$
set %arg to #11$    no, it must be a variable
$
quotarg '#'|        evaluate a quoted value argument
set %arg to #10$
set %f to 1$
$
quotarg '#|         evaluate a quoted value argument
set %arg to #10$
set %f to 1$
$
quotarg #| not a quoted value
$
aritharg #|         evaluate an arithmetic expression argument
set %tt to #10$
(#10)#17+-*/$                 seperate operators and operands
if %tt ne '#10' skip 2$        is it really an arithmetic expression?
#f9$                          no, quit
#f8$
arithfun #10$                 yes, see if it contains functions
%arg#16 #11#26 #24#26 #f3$
set %f to 1$
$
|     evaluate functions in arithmetic expressions
$
```

```
arithfun #(size #)#|          size function
evalfun(#10)(size)(#20)(#30)$
$
arithfun #(concat # #)#| concat function
evalfun(#10)(concat)(#20 #30)(#40)$
$
arithfun #(top #)#|           top of stack function
evalfun(#10)(top)(#20)(#30)$
$
arithfun #(stsize #)#|        stack size function
evalfun(#10)(stsize)(#20)(#30)$
$
arithfun #(stempty #)#|       stack empty function
evalfun(#10)(stempty)(#20)(#30)$
$
arithfun #(front #)#|         front of queue function
evalfun(#10)(front)(#20)(#30)$
$
arithfun #(back #)#|          back of queue function
evalfun(#10)(back)(#20)(#30)$
$
arithfun #(qsize #)#|         size of queue function
evalfun(#10)(qsize)(#20)(#30)$
$
arithfun #(qempty #)#|        queue emtpy function
evalfun(#10)(qempty)(#20)(#30)$
$
arithfun #|                no function in arithmetic expression
set %arg to #10$
$
evalfun(#)(#)(#)(#)|       evaluate function in arithmetic expression
arithfun #10$              1st and 4th parameters is rest of expression
%arg#16 #11#96$            2nd parameter is function name
arithfun #40$              3rd parameter is function argument(s)
#11#46$
funarg (#20 #30)$
number #11$
#90#11#40#26 #f3$
$
funarg (size #)|   evaluate a function argument
evalarg #10$
%arg#16 #11#16$
set %arg to #15$
set %f to 1$
$
funarg (concat '#' #)|
evalarg #20$
%arg#26$
```

```
set %arg to #10#21$
set %f to 1$
$
funarg (concat # #)|
evalarg #10$
%arg#16 #11#96$
evalarg #20$
set %arg to #90#11$
set %f to 1$
$
funarg (top #)|
if @#10 ne '' skip 4$
sterr #10<not initialized, returning null for top>$
set %arg to $
set %f to 1$
#f9$
@#10sp#96$
setv %arg to @#10#91$
set %f to 1$
$
funarg (stsize #)|
if @#10 ne '' skip 4$
sterr #10<not initialized, returning 0 for stsize>$
set %arg to 0$
set %f to 1$
#f9$
setv %arg to @#10sp$
set %f to 1$
$
funarg (stempty #)|
if @#10 ne '' skip 2$
sterr #10<not initialized, returning true for stempty>$
skip 1$
if @#10sp > 0 skip 2$
set %arg to true$
skip 1$
set %arg to false$
set %f to 1$
$
funarg (front #)|
if &#10 ne '' skip 4$
qerr #10<not initialized, returning null for front>$
set %arg to $
set %f to 1$
#f9$
&#10fr+1#96$
setv %arg to &#10#94$
set %f to 1$
```

```
$
funarg (back #)|
if &#10 ne '' skip 4$
qerr #10<not initialized, returning null for back>$
set %arg to $
set %f to 1$
#f9$
&#10ba-1#96$
setv %arg to &#10#94$
set %f to 1$
$
funarg (qsize #)|
if &#10 ne '' skip 4$
qerr #10<not initialized, returning 0 for qsize>$
set %arg to 0$
set %f to 1$
#f9$
setx %arg to &#10ba-&#10fr-1$
set %f to 1$
$
funarg (qempty #)|
if &#10 ne '' skip 2$
qerr #10<not initialized, returning true for qempty>$
skip 1$
if &#10ba-&#10fr-1 > 0 skip 2$
set %arg to true$
skip 1$
set %arg to false$
set %f to 1$
$
funarg #|  argument is not a function
$
arrayarg #!#|      evaluate an array variable argument
evalarg #20$
%arg#26$
setv %arg to #10!#21$
set %f to 1$
$
arrayarg #|        not an array variable
$
number #|        make sure argument is a number
#10#27+-$                  is there a + or -?
if '#10' eq '#20' skip 3$ no
if #25 = #15-1 skip 2$     yes, is it embedded or preceding?
set %arg to **#10**$       embedded, force expression error
#f9$
set %tn to #20$            preceding, remove it
skip 1$
```

```
#f8$
%tn#96$
#91#270123456789$
if '#20' ne '' skip 3$
#f8$
set %arg to #10$        ok, return good number
#f9$
set %arg to 1#101$      not a number, force an expression error
$
upop # #|       pop from a user stack (#10) onto variable (#20)
if @#10 ne '' skip 2$
sterr #10<not initialized, pop ignored>$
#f9$
if @#10sp > 0 skip 2$
sterr #10<pop on empty stack ignored>$
#f9$
@#10sp#96$
setv #20 to @#10#91$  set variable to stack element
set @#10#91 to $      clear stack element
setx @#10sp to @#10sp-1$    decrement stack pointer
$
sterr #<#>|       user stack error
** error on stack #10:  #20#f14$
$
inq# # #|         insert element (#30) onto front or back (#10)
if &#20 ne '' skip 2$       of a queue (#20)
qerr #20<not initialized, queue insertion ignored>$
#f9$
if &#20ba-&#20fr-1 < &#20 skip 2$
qerr #20<queue overflow, insertion ignored>$
#f9$
evalarg #30$
&#20#10#96$
setv &#20#91 to %arg$       insert the element
if '#10' eq 'fr' skip 2$    did we insert in front or back?
setx &#20#10 to &#20ba+1$   update back pointer
skip 1$
setx &#20#10 to &#20fr-1$   update front pointer
$
remq# # #|        remove element from front or back (#10) of queue (#20)
if &#20 ne '' skip 2$       and put it in variable (#30)
qerr #20<not initialized, queue removal ignored>$
#f9$
if &#20ba-&#20fr-1 > 0 skip 2$
qerr #20<removal from empty queue ignored>$
#f9$
if '#10' eq 'fr' skip 2$    did we remove in front or back?
setx &#20#10 to &#20ba-1$   update back pointer
```

```
skip 1$
setx &#20#10 to &#20fr+1$  update front pointer
&#20#10#96$
setv #30 to &#20#91$       put element in variable
set &#20#91 to $           remove element from queue
$
qerr #<#>|       queue error
** error on queue #10:  #20#f14$
$
if relexp('#' # #) skip #|   evaluate a relational expression and
evalarg #30$                 skip if true
%arg#96$
if relop (#10) #20 (#91) skip #40$
$
if relexp(# # #) skip #|
evalarg #10$
%arg#96 #91#16$
evalarg #30$
if relop (#10) #20 (#91) skip #40$
$
if relop (#) eq (#) skip #|   determine the relational operator
#f50$
$
if relop (#) ne (#) skip #|
#f51$
$
if relop (#) = (#) skip #|
#f60$
$
if relop (#) <> (#) skip #|
#f61$
$
if relop (#) < (#) skip #|
#f6-$
$
if relop (#) > (#) skip #|
#f6+$
$
if relop (#) <= (#) skip #|
if relop (#10) < (#20) skip #30+1$
#f60$
$
if relop (#) =< (#) skip #|
if relop (#10) <= (#20) skip #30$
$
if relop (#) => (#) skip #|
if relop (#10) > (#20) skip #30+1$
#f60$
```

```
$
if relop (#) >= (#) skip #|
if relop (#10) => (#20) skip #30$
$
|     if/then/else definitions
$
feoe|            find corresponding else or endif statement
30000#f7$
%ic#96 #91#96$
7#fi#90$
cif #70$
if %tf ne 't' skip 3$
setx %c to %c+1$
feoe$
skip 16$
cel #70$
if %tf ne 't' skip 5$
if %c = 0 skip 2$
setx %c to %c-1$
#f9$
#70$
#f9$
celd #70$
if %tf ne 't' skip 2$
if %c <> 0 skip 6$
#f9$
cen #70$
if %tf ne 't' skip 3$
if %c = 0 skip 1$
setx %c to %c-1$
#f9$
#f8$
$
eueoe|           execute until else or endif
30000#f7$
%ic#96 #91#96$
7#fi#90$
cel #70$
if %tf ne 't' skip 1$
#f9$
celd #70$
if %tf ne 't' skip 5$
push %c$
set %c to 0$
feoe$
pop %c$
#f9$
cen #70$
```

```
if %tf ne 't' skip 1$
#f9$
#70$
#f8$
$
|     check for the various types of if/then/else statements,
$     set %tf if found
cif  (if (#) then do)#|
set %tf to t$
$
cif  (if (#) then (#))#|
set %tf to t$
$
cif #|
set %tf to $
$
cel  (else (#))#|
set %tf to t$
$
cel #|
set %tf to $
$
celd  (else do)#|
set %tf to t$
$
celd #|
set %tf to $
$
cen  (endif)#|
set %tf to t$
$
cen #|
set %tf to $
$
cgoto  (goto #)#|
set %tf to t$
$
cgoto #|
set %tf to $
$
|     cbk definitions
$
x1#[#]|        route for good action and condition
if #15 > 0 skip 3$
**error no stub for condition or action#f14$
set err to 1$          set error flag
#f9$
if #15 < 38 skip 3$
```

```
set err to 1$
**error stub length > 38 chars.#f14$
#f9$
if fh > #15 skip 1$
set fh to #15$
#20#27,$
if #25 < 38 skip 3$
set err to 1$
**error entry length > 38 chars.#f14$
skip 3$
if bh => #25 skip 1$
set bh to #25$
#f8$
$
x1#|          bad condition or action
xe$
$
x1#[#|        bad condition or action
xe$
$
x1#]|         bad condition or action
xe$
$
xe|           error message
set err to 1$
**error unbalanced or missing brackets#f14$
$
```

APPENDIX 2.5

PROGRAMS USED TO TEST THE BARREL–F DEFINITION

The programs used to test the formal definition of Barrel–F are listed below.

Test 1 – A program taken from phase I of the Barrel/ASP Decision Table Entry, Translation, and Presentation system. It allows the user to enter the conditions and actions in the building of a decision table.

```
(zero c)
(zero a)
(zero fh)
(zero bh)
(setq ss '38)
(setq pn ')
(setq pk ')
(setq v ')
(setq sc 'make is [cord,reo,duesenberg]    ...  is a sample to follow)
(setq sa 'comm is [1%,5%,10%,variable]     ...  is a sample to follow)
(setq pfc 'please: a condition or "sample", "listc", "scut")
(setq pfa 'please: an action or "sample", "lista", "scut")
(ty ' )
(ty ' )
(loop1)
(if (pn ne ') then leave1)
(ty ' )
(ty 'please supply an external name for this table)
(readch '4' pn)
(if ((size pn) > '10) then do)
(ty '**error table name > 10 chars)
(setq pn ')
(endif)
(again1)
(loop2)
(if (pk eq 'e) then leave2)
(ty ' )
(ty 'please)
(ty '       a  "c" for entering conditions)
(ty '       an "a" for entering actions)
(ty '       an "e" to end input)
(ty ' )
```

```
(readch '4' pk)
(ty ' )
(if (pk ne 'e) then do)
(if (pk ne 'c) then do)
(if (pk ne 'a) then do)
(ty 'improper input information --- try again)
(setq pk ')
(endif)
(endif)
(if (pk ne ') then do)
(loop3)
(zero err)
(ty ' )
(if (pk eq 'c) then (ty pfc))
(else (ty pfa))
(ty ' )
(readch '4' v)
(if (v eq 'scut) then leave3)
(if (v eq 'sample) then do)
(if (pk eq 'c) then (ty sc))
(else (ty sa))
(else do)
(if (v eq 'listc) then (scn cond c 0))
(else do)
(if (v eq 'lista) then (scn act a 0))
(else do)
(cbk v)
(if (err eq '0) then do)
(if (pk eq 'c) then do)
(incr c)
(setq cond!c v)
(else do)
(incr a)
(setq act!a v)
(endif)
(endif)
(endif)
(endif)
(endif)
(again3)
(endif)
(endif)
(again2)
(setq xc (concat 'rem' c))
(setq xa (concat 'rem' a))
(setq cond!'0' xc)
(setq act!'0' xa)
(stop)
```

Test 2 – Test of the goto statement branching out of a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(goto lab1)
(ty 'bad1)
(if (a eq 'a) then leave1)
(ty 'bad2)
(again1)
(ty 'bad3)
(lab1:)
(ty 'ok2)
(stop)
```

Test 3 – Test of the goto statement branching into a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(goto lab1)
(loop2)
(ty 'bad1)
(lab1:)
(ty 'ok2)
(loop3)
(ty 'ok3)
(if (a eq 'a) then leave3)
(ty 'bad2)
(again3)
(ty 'ok4)
(if (a eq 'a) then leave2)
(ty 'bad3)
(again2)
(ty 'bad4)
(if (a eq 'a) then leave1)
(again1)
(ty 'bad5)
(stop)
```

Test 4 – Test of the goto statement branching into and then out of a loop.

```
(setq a 'a)|        testing goto - this should work
(goto lab1)
(loop1)
(ty 'bad1)
(lab1:)
(ty 'ok1)
(goto lab2)
```

```
(if (a eq 'a) then leave1)
(ty 'bad2)
(again1)
(ty 'bad3)
(lab2:)
(ty 'ok2)
(stop)
```

Test 5 – Test of the goto statement branching into a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(goto lab1)
(loop2)
(ty 'bad1)
(lab1:)
(ty 'ok2)
(if (a eq 'a) then leave2)
(ty 'bad2)
(again2)
(if (a eq 'a) then leave1)
(ty 'bad3)
(again1)
(ty 'ok3)
(stop)
```

Test 6 – Test of the goto statement branching within a loop.

```
(setq a 'a)|        testing goto - this should work
(loop1)
(ty 'ok1)
(loop2)
(ty 'ok2)
(goto lab1)
(ty 'bad1)
(lab1:)
(ty 'ok3)
(if (a eq 'a) then leave2)
(ty 'bad2)
(again2)
(ty 'ok4)
(if (a eq 'a) then leave1)
(ty 'bad3)
(again1)
(ty 'ok5)
(stop)
```

Test 7 – Test of the goto statement branching out of a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(loop2)
(ty 'ok2)
(goto lab1)
(ty 'bad1)
(if (a eq 'a) then leave2)
(ty 'bad2)
(again2)
(ty 'ok3)
(lab1:)
(ty 'ok4)
(if (a eq 'a) then leave1)
(ty 'bad4)
(again1)
(ty 'ok5)
(stop)
```

Test 8 – Test of the goto statement branching into a loop.

```
(setq a 'a)|        testing goto - this should fail
(ty 'ok1)
(goto lab1)
(loop1)
(ty 'bad1)
(lab1:)
(ty 'ok2)
(if (a eq 'a) then leave1)
(ty 'bad2)
(again1)
(ty 'ok3)
(stop)
```

Test 9 – Test of the goto statement branching within a loop.

```
(setq a 'a)|        testing goto - this should work
(loop1)
(ty 'ok1)
(goto lab1)
(ty 'bad1)
(lab1:)
(ty 'ok2)
(loop2)
(ty 'ok3)
(if (a eq 'a) then leave2)
(ty 'bad2)
(again2)
(ty 'ok4)
```

```
(if (a eq 'a) then leave1)
(ty 'bad3)
(again1)
(ty 'ok5)
(stop)
```

Test 10 – Test of the goto statement branching into a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(goto lab1)
(ty 'bad1)
(loop2)
(ty 'ok2)
(if (a eq 'a) then leave2)
(ty 'bad2)
(lab1:)
(ty 'ok3)
(again2)
(ty 'ok4)
(if (a eq 'a) then leave1)
(ty 'bad3)
(again1)
(ty 'ok5)
(stop)
```

Test 11 – Test of the goto statement branching within a loop.

```
(setq a 'a)|        testing goto - this should work
(loop1)
(ty 'ok1)
(if (a eq 'a) then (goto lab1))
(ty 'ok2)
(if (a eq 'b) then leave1)
(ty 'bad2)
(lab1:)
(ty 'ok3)
(setq a 'b)
(again1)
(ty 'ok4)
(stop)
```

Test 12 – Test of the goto statement branching into a loop.

```
(setq a 'a)|        testing goto - this should fail
(loop1)
(ty 'ok1)
(goto lab1)
```

```
(loop2)
(ty 'bad1)
(lab1:)
(ty 'ok2)
(loop3)
(ty 'ok3)
(goto lab2)
(if (a eq 'a) then leave3)
(again3)
(ty 'bad2)
(if (a eq 'a) then leave2)
(lab2:)
(ty 'ok4)
(again2)
(ty 'bad3)
(if (a eq 'a) then leave1)
(again1)
(ty 'bad4)
(stop)
```

Test 13 – Test of the goto statement branching out of an if/then statement.

```
(ty 'ok1)|                testing goto - this should work
(if ('a' eq 'a) then do)
(ty 'ok2)
(goto lab1)
(ty 'bad1)
(else do)
(ty 'bad2)
(endif)
(ty 'bad3)
(lab1:)
(ty 'ok3)
(stop)
```

Test 14 – Test of the goto statement branching within an if/then statement.

```
(ty 'ok1)|                testing goto - this should work
(if ('a' eq 'a) then do)
(ty 'ok2)
(goto lab1)
(ty 'bad1)
(else do)
(ty 'bad2)
(lab1:)
(ty 'ok3)
(endif)
(ty 'ok4)
(stop)
```

Test 15 – Test of the goto statement branching within an if/then statement.

```
(ty 'ok1)|                  testing goto - this should work
(if ('a' eq 'a) then do)
(ty 'ok2)
(goto lab1)
(ty 'bad1)
(lab1:)
(ty 'ok3)
(else do)
(ty 'bad2)
(endif)
(ty 'ok4)
(stop)
```

Test 16 – Test of the goto statement branching into an if/then statement.

```
(ty 'ok1)|                  testing goto - this should work
(goto lab1)
(ty 'bad1)
(if ('a' eq 'a) then do)
(ty 'bad2)
(lab1:)
(ty 'ok2)
(else do)
(ty 'ok3)
(endif)
(ty 'ok4)
(stop)
```

Test 17 – Test of the goto statement with the if/then statement.

```
(if ('a' eq 'a) then do)|          testing goto - this should work
(goto lab1)
(endif)
(ty 'bad1)
(lab1:)
(ty 'ok1)
(if ('a' eq 'a) then (goto lab2))
(ty 'bad2)
(lab2:)
(ty 'ok2)
(if ('a' ne 'a) then (ty 'bad3a))
(else (goto lab3))
(ty 'bad3b)
(lab3:)
(ty 'ok3)
(stop)
```

Test 18 – Test of the setq, incr, decr, bump, and zero statements.

```
(setq a '1)
(incr a)
(ty a)
(decr a)
(ty a)
(bump a 2*a+3)
(ty a)
(setq b a)
(ty b)
(zero a)
(ty a)
(stop)
```

Test 19 – A program to find the factorial of n.

```
(readch '4' n)
(setq i '1)
(setq fact '1)
(if (n <> '0) then do)
(loop1)
(if (i = n) then leave1)
(incr i)
(setq fact fact*i)
(again1)
(endif)
(ty 'the factorial of ,n,' is ,fact)
(stop)
```

Test 20 – A program to test the size function.

```
(setq n 'abc)
(setq a (size n))
(ty a)
(setq a (size 'abcd))
(ty a)
(setq b '1)
(setq c '12)
(setq a (size b+c))
(ty a)
(setq a (size b))
(ty a)
(ty 'hello,(size c*c),a)
(setq a 2*(size 'abc)+1)
(ty a)
(ty (size (size 'abcdefghij)))
(stop)
```

Test 21 – A program to test the plink operator.

```
(setq arr!'0' 'hello)
(setq c '1)
(setq arr!c c+1)
(readch '4' arr!c+1)
(if (arr!'0' eq arr!'1') then (ty 'bad news))
(else (ty 'good news))
(zero b!'10')
(loop1)
(ty arr!b!'10')
(if (b!'10' = '2) then leave1)
(incr b!'10')
(again1)
(setq arr!'# one' 'any value)
(ty arr!'# one)
(setq b '# one)
(ty arr!b)
(stop)
```

Test 22 – A program to test the concat function.

```
(setq a (concat '1st half' '2nd half))
(ty a)
(setq a (concat '1st half' '2nd half'))
(ty a)
(setq b '2nd half)
(setq a (concat '1st half' b))
(ty a)
(setq b '1st half)
(setq a (concat b '2nd half))
(ty a)
(ty (concat 1+3*2 (concat '7' (size '1234567'))))
(stop)
```

Test 23 – A program to test the cbk statement.

```
(zero c)
(zero fh)
(zero bh)
(setq tab!'0' 't < 0 [])
(setq tab!'1' 'make is [cord,reo,dues])
(setq tab!'2' 'condition is good,bad])
(setq tab!'3' 'comm is [1%,5%)
(setq tab!'4' '[needed,not needed])
(setq tab!'5' 'this stub is going to be much much too long [1,2])
(setq tab!'6' 'stub [this entry is going to be much too long,2])
(setq tab!'7' 'managers ok is [needed, not needed])
(loop1)
```

```
(cbk tab!c)
(ty fh,bh)
(if (c = '7) then leave1)
(incr c)
(again1)
(stop)
```

Test 24 – A program to test the scn statement.

```
(setq cond!'0' 'rem3)
(setq cond!'1' 'make is [cord,reo,dues])
(setq cond!'2' 'cond is [good,bad])
(setq cond!'3' 'comm is [1%,5%])
(setq cond!'4' 'shopwork is [needed,not needed])
(setq c '4)
(scn cond c 0)
(scn cond c 1)
(stop)
```

Test 25 – A program to test setq and bump with quoted literal values.

```
(setq a '1)
(bump a 'abc)|        this doesn't make sense
(ty a)
(setq a '1)
(bump a '12-10)|      expressions cannot be quoted
(ty a)
(bump a 12-10)
(ty a)
(setq 1 '2)
(setq a 1)|           numbers must be quoted or they are identifiers
(ty a)
(stop)
```

Test 26 – A program to test the loop statement.

```
(ty 'hello)
(zero zz)
(zero yyy)
(loop1)
(incr zz)
(if (zz = '4) then leave1)
(incr yyy)
(again1)
(ty zz)
(ty yyy)
(stop)
```

Test 27 – A program to test the execute statement.

```
(setq a '1)
(ty a)
(readch '4' code)|  input should be (setq a '2)
(ty code)
(execute code)
(ty a)
(stop)
```

Test 28 – A program to test the execute statement.

```
(setq a '1)
(ty a)
(readch '4' code)|  input should be (setq b '2)
(ty code)
(execute code)
(readch '4' code)|  input should be (ty b)
(ty code)
(execute code)
(stop)
```

Test 29 – A program to test the ty statement.

```
(setq s 'yucky)
(setq t 'pooh)
(ty 's is ,s,' t is ,t)
(ty s,'is s)
(ty '(a,b))
(ty 'hello there)
(ty '(a b))
(ty 's = ,s)
(setq a '1,&%2)
(ty 'a is ,a)
(stop)
```

Test 30 – A program to test the if/then/else statement.

```
(readch '4' i)
(ty 'test 1)
(if (i = '0) then (ty '1 =))
(ty 'test 2)
(if (i = '0) then (ty '2 =))
(else (ty '2 <>))
(ty 'test 3)
(if (i = '0) then (ty '3 =))
(else do)
(ty '3 <>)
(endif)
(ty 'test 4)
(if (i = '0) then do)
```

```
(ty '4 =)
(endif)
(ty 'test 5)
(if (i = '0) then do)
(ty '5 =)
(else (ty '5 <>))
(ty 'test 6)
(if (i = '0) then do)
(ty '6 =)
(else do)
(ty '6 <>)
(endif)
(ty 'through now)
(stop)
```

Test 31 – A program to test the relational operators.

```
(readch '4' a)
(readch '4' b)
(if (a eq b) then (ty 'a eq b))
(if (a ne b) then (ty 'a ne b))
(readch '4' a)
(readch '4' b)
(if (a = b) then (ty 'a = b))
(if (a <> b) then (ty 'a <> b))
(if (a < b) then (ty 'a < b))
(if (a > b) then (ty 'a > b))
(if (a <= b) then (ty 'a <= b))
(if (a =< b) then (ty 'a =< b))
(if (a >= b) then (ty 'a >= b))
(if (a => b) then (ty 'a => b))
(stop)
```

Test 32 – A program to test the relational expressions.

```
(setq a 'abc)
(if (a eq 'abc) then (ty '1 ok))
(setq a 1+1)
(if (a = 1+1) then (ty '2 ok))
(if (a < '3) then (ty '3 ok))
(if (a > '1) then (ty '4 ok))
(if (a >= '2) then (ty '5 ok))
(if (a =< a+1) then (ty '6 ok))
(ty 'through now)
(stop)
```

Test 33 – A program to find the factorial of n.

```
(more:)
(readch '4' n)
```

```
(if (n = '100) then (goto fin))
(setq i '1)
(setq fact '1)
(if (n = '0) then (goto out))
(over:)
(if (i = n) then (goto out))
(incr i)
(setq fact fact*i)
(goto over)
(out:)
(ty n)
(ty fact)
(goto more)
(fin:)
(stop)
```

Test 34 – The infamous exec program used to interactively execute commands (a command line interpreter).

```
(ty 'we permit 10 executions only)
(ty '  all submitted code should start in column 2)
(setq i '1)
(loop1)
(ty 'send:)
(readch '4' code)
(execute code)
(incr i)
(if (i > '10) then leave1)
(again1)
(stop)
```

Test 35 – A program to test the qinit, inqfront, and remqfront statements.

```
(qinit q '2)
(inqfront q 'frog)
(remqfront q var)
(ty var)
(stop)
```

Test 36 – A program to test the qcopy statement.

```
(qinit q1 '3)
(qinit q2 '2)
(inqfront q1 'lilly)
(inqback q1 'pad)
(qcopy q1 to q2)
(remqfront q2 var)
(ty var)
```

```
(remqfront q2 var)
(ty var)
(stop)
```

Test 37 – A program to test the front of queue function.

```
(qinit q '2)
(inqfront q 'lilly)
(ty (front q))
(inqback q 'pad)
(setq frog (front q))
(ty frog)
(remqfront q frog)
(ty frog,(front q))
(stop)
```

Test 38 – A program to test the back of queue function.

```
(qinit q '2)
(inqfront q 'lilly)
(ty (back q))
(inqback q 'pad)
(setq frog (back q))
(ty frog)
(remqfront q frog)
(ty frog,(back q))
(stop)
```

Test 39 – A program to test the queue size function.

```
(qinit q '2)
(ty (qsize q))
(inqfront q 'lilly)
(ty (qsize q))
(inqback q 'pad)
(setq frog (qsize q))
(ty frog)
(remqfront q frog)
(if ((qsize q)+1 = '2) then (ty 'ok1))
(else (ty 'whoops))
(stop)
```

Test 40 – A program to test the queue empty function.

```
(qinit q '2)
(if ((qempty q) eq 'true) then (ty 'ok1))
(else (ty 'whoops1))
(inqfront q 'lilly)
(if ((qempty q) eq 'false) then (ty 'ok2))
```

```
(else (ty 'whoops2))
(inqback q 'pad)
(if ((qempty q) eq 'true) then (ty 'whoops3))
(else (ty 'ok3))
(qinit q '3)
(if ((qempty q) eq 'true) then (ty 'ok4))
(else (ty 'whoops4))
(stop)
```

Test 41 – A program to test the queue empty function.

```
(qinit q '2)
(ty (qempty q))
(inqfront q 'lilly)
(setq frog 1+(qempty q))
(ty frog)
(inqback q (qempty q))
(remqback q frog)
(ty frog)
(stop)
```

Test 42 – A program to test the qinit, inqback, and remqback statements.

```
(qinit q '2)
(inqback q 'frog)
(remqback q var)
(ty var)
(stop)
```

Test 43 – A program to test the inqfront and remqfront statements.

```
(inqfront q 'frog)|    this fails
(remqfront q wart)
(stop)
```

Test 44 – A program to test the inqback and remqback statements.

```
(inqback q 'frog)|    this fails
(remqback q wart)
(stop)
```

Test 45 – A program to test the overflowing and unerflowing of inqfront and remqfront.

```
(qinit q '2)|          this overflows
(setq var 'frog)
(inqfront q var)
(inqfront q (concat 'lilly' 'pad))
(inqfront q 5*5)
```

```
(remqfront q var1)|     this underflows
(ty var1)
(remqfront q var1)
(ty var1)
(remqfront q var1)
(ty var1)
(stop)
```

Test 46 – A program to test the overflowing and unerflowing of inqback and remqback.

```
(qinit q '2)|           this overflows
(setq var 'frog)
(inqback q var)
(inqback q (concat 'lilly' 'pad))
(inqback q 5*5)
(remqback q var1)|     this underflows
(ty var1)
(remqback q var1)
(ty var1)
(remqback q var1)
(ty var1)
(stop)
```

Test 47 – A program to test the overflowing and unerflowing of inqfront and remqback.

```
(qinit q '2)|           this overflows
(setq var 'frog)
(inqfront q var)
(inqfront q (concat 'lilly' 'pad))
(inqfront q 5*5)
(remqback q var1)|     this underflows
(ty var1)
(remqback q var1)
(ty var1)
(remqback q var1)
(ty var1)
(stop)
```

Test 48 – A program to test the overflowing and unerflowing of inqback and remqfront.

```
(qinit q '2)|           this overflows
(setq var 'frog)
(inqback q var)
(inqback q (concat 'lilly' 'pad))
```

```
(inqback q 5*5)
(remqfront q var1)|     this underflows
(ty var1)
(remqfront q var1)
(ty var1)
(remqfront q var1)
(ty var1)
(stop)
```

Test 49 – A program to test the qty statement.

```
(qinit q '3)
(setq var 'frog)
(inqback q var)
(inqback q (concat 'lilly' 'pad))
(inqback q 5*5)
(qty q)
(remqback q var1)
(qty q)
(remqback q var1)
(qty q)
(remqback q var1)
(qty q)
(stop)
```

Test 50 – A program to test the stinit, push, and pop statements.

```
(stinit st '2)
(push st 'frog)
(pop st var)
(ty var)
(stop)
```

Test 51 – A program to test the push and pop statements.

```
(push st 'frog)|        this fails
(pop st wart)
(stop)
```

Test 52 – A program to test the overflowing and underflowing of the push and pop statements.

```
(stinit st '2)|       this overflows
(setq var 'frog)
(push st var)
(push st (concat 'lilly' 'pad))
(push st 5*5)
(pop st var1)|        this underflows
(ty var1)
```

```
(pop st var1)
(ty var1)
(pop st var1)
(ty var1)
(stop)
```

Test 53 – A program to test the stcopy statement.

```
(stinit st1 '3)
(stinit st2 '2)
(push st1 'lilly)
(push st1 'pad)
(stcopy st1 to st2)
(pop st2 var)
(ty var)
(pop st2 var)
(ty var)
(stop)
```

Test 54 – A program to test the top of stack function.

```
(stinit st '2)
(push st 'lilly)
(ty (top st))
(push st 'pad)
(setq frog (top st))
(ty frog)
(pop st frog)
(ty frog,(top st))
(stop)
```

Test 55 – A program to test the stack size function.

```
(stinit st '2)
(ty (stsize st))
(push st 'lilly)
(ty (stsize st))
(push st 'pad)
(setq frog (stsize st))
(ty frog)
(pop st frog)
(if ((stsize st)+1 = '2) then (ty 'ok1))
(else (ty 'whoops))
(stop)
```

Test 56 – A program to test the stack empty function.

```
(stinit st '2)
(if ((stempty st) eq 'true) then (ty 'ok1))
```

```
(else (ty 'whoops1))
(push st 'lilly)
(if ((stempty st) eq 'false) then (ty 'ok2))
(else (ty 'whoops2))
(push st 'pad)
(if ((stempty st) eq 'true) then (ty 'whoops3))
(else (ty 'ok3))
(stinit st '3)
(if ((stempty st) eq 'true) then (ty 'ok4))
(else (ty 'whoops4))
(stop)
```

Test 57 – A program to test the stack empty function.

```
(stinit st '2)
(ty (stempty st))
(push st 'lilly)
(setq frog 1+(stempty st))
(ty frog)
(push st (stempty st))
(pop st frog)
(ty frog)
(stop)
```

APPENDIX 2.6

INFORMAL DESCRIPTION OF ASP CODE BODIES

Informal description of the ASP code bodies, a mixture of low–level constructs and high–level Barrel–F statements. The goto, loop, and if statements of Barrel–F are not included since they cannot be used in code bodies; however, their function is duplicated by processor functions. Keywords are lower case; non–terminals are upper case. A "general case" example is followed by a specific example.

1. setq – assigns a value to a variable
   example: (setq VARIABLE EXP)
   (setq var 'some text)
   (setq var a+3*b)
   (setq var anothervar)

2. zero – sets the value of the variable to zero
   example: (zero VARIABLE)
   (zero var)

3. bump – adds the value of the expression to the value of the variable and makes that the new value of the variable
   example: (bump VARIABLE EXP)
   (bump var 2*c)

4. incr – increment the value of the variable by one
   example: (incr VARIABLE)
   (incr var)

5. decr – decrement the value of the variable by one
   example: (decr VARIABLE)
   (decr var)

6. execute – execute the value of the variable as if it were a statement
   example: (execute VARIABLE)
   (execute var)

7. readch – get an input value from channel 4 (as defined by ASP) and make it the new value of the variable
   example: (readch '4' VARIABLE)
   (readch '4' var)

8. ty – output the value of the variable or the quoted value; if there is more than one argument (separated by commas) concatenate them before output; arithmetic expressions are not supported
   example: (ty EXP[,EXP...])
   (ty 'the value of a is,a)

9. cbk – determine if the value of the variable is a valid condition or action (i.e. suitable for inclusion in a codebook as defined for the Barrel/ASP Decision Table Entry, Translation, and Presentation System); if not valid, print an error message and set the value of the variable "err" to 1; if valid, set the value of the variable "fh" to the length of the stub and set the value of the variable "bh" to the length of the longest entry (fh and bh are only set if their values are less than the values just found, i.e. if the values are bigger than any other found so far in the program
   example: (cbk VARIABLE)
   (cbk var)

10. scn – output the contents of an array; the array name is given by the first variable, the starting point (either the zero or first element) is given by the integer, and the ending point is given by the value of the second variable
   example: (scn VARIABLE VARIABLE LIT)
   (scn array length 1)

11. stinit – initialize or clear a stack giving it the specified number of elements
   example: (stinit STACK EXP)
   (stinit ast '20)

12. push – push a value onto a stack
   example: (push STACK EXP)
   (push ast 'a value)

13. pop – remove a value from a stack and make it the new value of a variable
   example: (pop STACK VARIABLE)
   (pop ast var)

14. stcopy – copy the values of one stack to another stack destroying the previous contents of the target stack

example: (stcopy STACK1 to STACK2)
(stcopy thisstack to thatstack)

15. qinit – initialize or clear a queue giving it the specified number of elements
example: (qinit QUEUE EXP)
(qinit aqu '20)

16. inqfront – insert a value onto the front of a queue
example: (inqfront QUEUE EXP)
(inqfront aqu a*b)

17. inqback – insert a value onto the back of a queue
example: (inqback QUEUE EXP)
(inqback aqu a*b)

18. remqfront – remove a value from the front of a queue and make it the new value of a variable
example: (remqfront QUEUE VARIABLE)
(remqfront aqu var)

19. remqback – remove a value from the back of a queue and make it the new value of a variable
example: (remqback QUEUE VARIABLE)
(remqback aqu var)

20. qty – output the contents of a queue from front to back
example: (qty QUEUE)
(qty aqu)

21. qcopy – copy the values of one queue to another queue destroying the previous contents of the target queue
example: (qcopy QUEUE1 to QUEUE2)
(qcopy thisq to thatq)

22. stop – stop execution of the program
example: (stop)

23. size – function; returns the precision if the value of the argument is a number or returns the length if the value of the argument is a character string
example: (size EXP)
(setq a (size 'this string))
(setq b (size a*b+c))

24. concat – function; returns the concatenation of the character string values of the two arguments
    example: (concat EXP EXP)
    (setq a (concat a 'this string))

25. top – function; returns the value currently on top of the stack without popping it
    example: (top STACK)
    (setq var (top st1))

26. stsize – function; returns the number of values currently on the stack
    example: (stsize STACK)
    (ty (stsize st1),' elements)

27. stempty – function; returns true if the stack has no values on it and returns false otherwise
    example: (stempty STACK)
    (if ((stempty st1) eq 'true) then (goto empty))

28. front – function; returns the value currently on the front of the queue without removing it
    example: (front QUEUE)
    (ty (front aqu))

29. back – function; returns the value currently on the back of the queue without removing it
    example: (back QUEUE)
    (ty (back aqu))

30. qsize – function; returns the number of values currently on the queue
    example: (qsize QUEUE)
    (setq a (qsize aqu)+3)

31. qempty – function; returns true if the queue has no values on it and returns false otherwise
    example: (qempty QUEUE)
    (if ((qempty aqu) eq 'true) then (goto empty))

32. ! – plink operator; allows specification of the index of an array variable (the index can evaluate to an integer or a character string)
    example: (setq VARIABLE!EXP EXP)
    (setq arr!'3' 'a value)
    (ty 'the name is: ,arr!'name')

NOTE: in the following descriptions "constructed line" is the line that is being built from the characters, parameter transformations, and processor functions contained in it; the constructed line, when built, is submitted for execution if it is a previously defined statement or output to channel 3 if not.

NOTE: in the following description of the parameter transformations P stands for the parameter number (1–9)

33. #P0 – parameter transformation 0; copy the value of the parameter to the constructed line
    example: parameter 1 is #10

34. #P1 – treat the value of the parameter as a variable name and copy its value to the constructed line
    example: memory value of parameter 1 is #11

35. #P4 – treat the value of the parameter as an arithmetic expression and evaluate it and copy the value to the constructed line
    example: 1+2 is #14

36. #P5 – copy the length of the value of the parameter to the constructed line
    example: length of parameter 1 is #15

37. #P6 – treat the value of the parameter as a variable name and make its value the value of the constructed line
    example: new value for parameter 1#16

38. #P7 – looping construct; the value of the parameter is saved for later restoration, the character(s) which follow #P7 are defined as break characters, if the constructed line (the characters before #P7) is enclosed in parentheses the outermost pair is removed, the longest balanced string (balanced with respect to parentheses) which begins at the beginning of the constructed line and contains no break characters not enclosed in parentheses is made the value of the parameter, the string and the break character which follows it are removed from the constructed line, processing of the code body continues until processor function #F8 is encountered at which time control returns to the line containing the #P7 and the process repeats itself, if the constructed line of the #P7 is null when a corresponding #F8 is encountered the looping terminates and the original value of the parameter is restored and execution of the code body continues with the element following the #F8, if there are no break characters then the first character of the

constructed line is made the value of the parameter and that character is deleted from the constructed line and execution continues as above
example: a,bc,d#17,
(ty #10)
#f8

NOTE: the following are descriptions of processor functions; the F is any non-digit except # or $

39. #F0 - terminate processing
example: #f0

40. #F14 - output the constructed line on channel 4 (as defined by the ASP implementation)
example: hello#f14

41. #F3 - the value of parameter 1 is treated as a variable name which is given as its new value the value of parameter 2
example: var#16 val#26 #f3

42. #F4 - the value of parameter 1 is treated as arithmetic expression and is evaluated and a number of lines equal to this value are skipped
example: n+1#16 #f4

43. #F5k - if k is 0 and the values of parameters 1 and 2 are equal then the value of parameter 3 is treated as an arithmetic expression and is evaluated and a number of lines equal to this value are skipped, if k is 1 and the values of parameters 1 and 2 are not equal then the value of parameter 3 is treated as an arithmetic expression and is evaluated and a number of lines equal to this value are skipped
example: a#16 b#26 n+1#36 #f51

44. #F6k - the values of parameters 1 and 2 are treated as arithmetic expressions and are evaluated, if k is 0 the values are tested for equality, if k is 1 the values are tested for non-equality, if k is + the values are tested to see if the value obtained from the evaluation of parameter 1 is greater than the value obtained from the evaluation of parameter 2, if k is - the values are tested to see if the value obtained from the evaluation of parameter 1 is less than the value obtained from the evaluation of parameter 2, if the test is true then parameter 3 is treated as an arithmetic expression and is evaluated and a number of lines equal to this value are skipped
example: 2#16 1+1#26 n+1#36 #f60

45. #F7 – looping construct; the constructed line is treated as an arithmetic expression and is evaluated, execution is continued with the next element following #F7, when a corresponding #F8 is encountered control of execution is returned to the element following the #F7, this is repeated a number of times equal to the value obtained from the evaluation of the constructed line
example: n+1#f7

46. #F8 – signifies the end of a looping construct (see #P7 and #F7 above)
example: #f8

47. #F9 – escape from processing the code body
example: #f9

48. #Fi4 – treat the element immediately preceding #Fi4 as a parameter number, get an input value from channel 4 (as defined by the ASP implementation) and make it the new value of the parameter
example: 5#fi4

Definition of non-keywords used in examples above:

VARIABLE — a variable name which can consist of any sequence of characters which are balanced with respect to parentheses

EXP — can be an arithmetic expression
example: 2*(a+1)
or a function call
example: (size box)
or a variable name
example: netpay
or a quoted value (a literal)
example: 'the rain in Spain
NOTE: a quoted value is delimited by the closing parenthesis when used in a setq statement and by a comma or the closing parenthesis when used in a ty statement
NOTE: an arithmetic expression can involve the four arithmetic operations +, –, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can

serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)

BOOLOP — a boolean operator; can be any of:
eq (string equality)
ne (string inequality)
= (equal)
< > (not equal)
< (less than)
> (greater than)
< = (less than or equal)
= < (less than or equal)
> = (greater than or equal)
= > (greater than or equal)
NOTE: eq and ne assume their arguments are strings and the other relational operators assume their arguments are integers

STATEMENT — can be any single statement

STATEMENTS — can be any sequence of zero or more statements
NOTE: each statement must fit on one line (usually 80 characters but implementation dependent) so all non-keywords have an implied limit to their size

STACK — the name of a stack; see VARIABLE

QUEUE — the name of a queue; see VARIABLE

LIT — a literal constant value

APPENDIX 2.7

FORMAL DEFINITION OF ASP CODE BODIES

Below is the formal definition of the code bodies of ASP, i.e., the lines that can be included in a code body. Defined are the ASP processor functions, parameter transformations, and most of the Barrel–F statements which can be called from a code body.

We first present a Prolog interface to the formal definition which makes its execution very simple. The user simply enters "go(file)." where file is the name of a file which contains an ASP code body (without the template). Each of the three parts of the definition (listed below) are called in turn: lexeme – the lexical syntax, morpheme – the syntax, and sememe – the semantics.

```
                    /*********************/
                    /*** main program ***/
                    /*********************/

/*** define the top level of the Barrel-F definition ***/
/*** the major predicates are lexemes, morpheme, and sememe ***/
go(File) :- see(File), read_in(Text), seen,
            lexemes(Tokens, Text, []), !,
            write('Tokens = '), pp(Tokens, 50, 9), nl, nl,
            morpheme([Tree | Mem], Tokens, []), !,
            write('Tree = '), pp(Tree, 50, 7), nl, nl,
            prettylist(Mem),
            write('Mem before sememe = '), pp(Mem, 50, 20), nl, nl,
    write('Enter your input in list form and end it with a period: '),
            read(Input), nl,
            uglylist(Mem, Mem1), !,
            sememe(Tree, state(Mem1, [], Input, Output, Chan3, noloop,
                   ok),
                         state(M1, Cont, I1, O1, C3a, L1, Result)),
            prettylist(M1),
            write('Mem after sememe = '), pp(M1, 50, 19), nl, nl,
            write('Input = '), pp(Input, 50, 8), nl, nl,
            prettylist(Output),
```

```
            write('Output = '), pp(Output, 50, 9), nl, nl,
            prettylist(Chan3),
            write('Channel 3 = '), pp(Chan3, 50, 12), nl, nl,
            write('Result = '), write(Result), nl.

/*** read each character from a file into a list of characters ***/
read_in([W | Ws]) :- get0(W), not checkeof(W), read_in(Ws), !.
read_in([]).
checkeof(26).  /* check for end of file */

/*** get rid of uninstantiated variable at the tail of a list ***/
prettylist([]).
prettylist([Head | Tail]) :- var(Tail), Tail = [].
prettylist([Head | Tail]) :- prettylist(Tail).

/*** put uninstantiated variable at tail of a list ***/
/*** (opposite of prettylist) ***/
uglylist(List, []) :- isnull(List).
uglylist(List, Newlist) :- not isnull(List), putvar(List, Newlist).
putvar([X], [X | _]).
putvar([X | Y], [X | Z]) :- putvar(Y, Z).

/* Useful for printing long lists (more than 80 characters). */
/* It inserts newlines after every CPL characters and indents */
/* each line Sc characters. It uses file @@@ */
pp(List, CPL, Sc) :- not exists('@@@'), tell('@@@'),
                     write(List), told, see('@@@'),
                     pp1(1, CPL, Sc), seen, system("rm @@@").
pp(List, CPL, Sc) :- exists('@@@'), write(List).
pp1(Count, CPL, Sc) :- pp2(Count, CPL, Fl), (Fl = e;
                       nl, tab(Sc), pp1(Count, CPL, Sc)).
pp2(Count, CPL, Fl) :- CPL < 2, pp2(Count, 3, Fl).
pp2(Count, CPL, Fl) :- Count < CPL, get0(CH), (CH = 26, Fl = e;
                       put(CH), NewC is Count+1, pp2(NewC, CPL, Fl)).
pp2(Count, CPL, Fl) :- Count = CPL, Fl = n.

/*** consult the other files needed for the Barrel-F definition ***/
:- [lexemes, morpheme, syncon, sememe].

                  /*********************************/
                 /*****   lexemes portion   *****/
                /*********************************/

/*** produce a list of tokens from the list of characters ***/
lexemes([X|Y]) --> lexeme(X), lexemes(Y).
lexemes([]) --> [].

/*** a lexeme is a list of tokens for one line of the program ***/
lexeme([]) --> [CH], {isnewline(CH)}.
lexeme([]) --> comment.
lexeme([X|Y]) --> token(X), lexeme(Y).
```

```
/*** get rid of comments ***/
comment --> [CH], {iseol(CH)}, restofcomment.
restofcomment --> [CH], {not isnewline(CH)}, restofcomment.
restofcomment --> [CH], {isnewline(CH)}.

/*** a token is a possible identifier ***/
token(id(Word)) --> word(W), {list(Word, W)}.
/* or a processor function (pf) */
/* or a parameter transformation (pt) */
token(PFPT) --> pfpt(PFPT).
/* or any other character */
token(Other) --> [CH], {list(Other, [CH])}.

/*** build a word from alphanumeric characters ***/
word([First|Rest]) --> char(First), word(Rest).
/* quit when we get to a non-alphanumeric character */
word([Last]), [Next] --> char(Last), notchar(Next).

char(CH) --> [CH], {ischar(CH)} .
notchar(CH) --> [CH], {not ischar(CH)}.

/*** '#' delimits a pt or pf and '$' signals end-of-line ***/
/*** we can use them as normal characters by using a '#' ***/
/*** before it ***/
pfpt('#') --> [35], [35].
pfpt('$') --> [35], [36].
/* otherwise a '#' means we have a pt or pf */
pfpt(['#', Any, CH]) --> [35], [Any1], [CH1], {notequal(Any1, 35),
                         notequal(Any1, 36), list(Any, [Any1])},
                         sppfpt(Any, CH1, L), {list(CH, L)}.

/*** many pt/pf's require special handling of the ***/
/*** characters following them ***/
sppfpt(Any, 54, [54]) --> {integer(Any)}, pfpt(_).
sppfpt(Any, 54, [54]) --> {integer(Any)}, nextch(_).
sppfpt(Any, 49, [49, Chan]) --> {not integer(Any)}, nextch(Chan),
                                nextch(Rewind), nextch(_).
sppfpt(Any, 49, [49, Chan]) --> {not integer(Any)}, nextch(Chan).
sppfpt(Any, 51, [51]) --> {not integer(Any)}, nextch(_).
sppfpt(Any, 52, [52]) --> {not integer(Any)}, nextch(_).
sppfpt(Any, 53, [53, K]) --> {not integer(Any)}, nextch(K), nextch(_).
sppfpt(Any, 53, [53, K]), [CH] --> {not integer(Any)}, nextch(K), [CH],
                                   {(isnewline(CH); iseol(CH))}.
sppfpt(Any, 54, [54, K]) --> {not integer(Any)}, nextch(K), nextch(_).
sppfpt(Any, 54, [54, K]), [CH] --> {not integer(Any)}, nextch(K), [CH],
                                   {(isnewline(CH); iseol(CH))}.
sppfpt(Any, 55, [55]) --> {not integer(Any)}, nextch(_).
sppfpt(Any, 56, [56]) --> {not integer(Any)}, nextch(_).
sppfpt(Any, 105, [105, Chan]) --> {not integer(Any)}, nextch(Chan),
                                  nextch(_).
```

```
sppfpt(Any, 105, [105, Chan]), [CH] --> {not integer(Any)},
                                        nextch(Chan), [CH],
                                        {(isnewline(CH); iseol(CH))}.
/*** handle all other pt/pf's ***/
sppfpt(Any, X, [X]) --> [].

/*** get the next character as long as it's not a ***/
/*** newline or end-of-line ***/
nextch(CH) --> [CH], {not isnewline(CH), not iseol(CH)}.

                  /*******************************/
                 /*****   syntax portion   *****/
                /*******************************/

/*** build a Tree of abstract syntax statements and an ***/
/*** Environment of variables and their values ***/
morpheme([Tree | Env]) --> stmtrain(Env, Tree), [[]].

/*** process the statements of the program ***/
stmtrain(Env, [Sem | Seml]) --> statement(Env, Sem),
                                (stmtrain(Env, Seml);
                                 {Seml = []}).

/*** process an individual statement ***/
statement(Env, Sem) --> [Line], /* one line per statement */
                        ({setqstm(Env, Sem, Line, []);
                          zerostm(Env, Sem, Line, []);
                          bumpstm(Env, Sem, Line, []);
                          incrstm(Env, Sem, Line, []);
                          decrstm(Env, Sem, Line, []);
                          executestm(Env, Sem, Line, []);
                          transputstm(Env, Sem, Line, []);
                          cbkstm(Env, Sem, Line, []);
                          scnstm(Env, Sem, Line, []);
                          stackstm(Env, Sem, Line, []);
                          queuestm(Env, Sem, Line, []);
                          stopstm(Sem, Line, [])});
                       constline(Env, Sem).

/*** assignment statement ***/
setqstm(Env, setq(Tag, Exp)) --> begofstm(setq),
                                 identifier(Env, Tag, ' '), [' '],
                                 exp(Env, Exp, ')'), [')'].

/*** set variable to zero statement ***/
zerostm(Env, setq(Tag, val(0))) --> begofstm(zero),
                                    identifier(Env, Tag, ')'), [')'].

/*** bump a variable by the value of an arithmetic expression ***/
bumpstm(Env, setq(Tag, plus(deref(Tag), Exp))) --> begofstm(bump),
```

```
            identifier(Env, Tag, ' '), [' '], exp(Env, Exp, ')'),
            [')'].

/*** increment the value of a variable ***/
incrstm(Env, setq(Tag, plus(deref(Tag), val(1)))) --> begofstm(incr),
            identifier(Env, Tag, ')'), [')'].

/*** decrement the value of a variable ***/
decrstm(Env, setq(Tag, minus(deref(Tag), val(1)))) --> begofstm(decr),
            identifier(Env, Tag, ')'), [')'].

/*** execute the value of a variable ***/
executestm(Env, execute(Tag)) --> begofstm(execute),
                                  identifier(Env, Tag, ')'), [')'].

/*** input and output statements ***/
transputstm(Env, output(Exp)) --> begofstm(ty),
                                  outexp(Env, Exp), {!}, [')'].
transputstm(Env, input(Tag)) --> begofstm(readch), {genquote(Quote)},
                                 [Quote], [id(4)], [Quote], [' '],
                                 identifier(Env, Tag, ')'), [')'].

/*** check for valid condition or action in the ***/
/*** codebook of a decision table ***/
cbkstm(Env, cbk(Tag)) --> begofstm(cbk), identifier(Env, Tag, ')'),
                          [')'], {declare(err, undef, Env)}.

/*** output the contents of an array ***/
scnstm(Env, scn(id(Tag), id(End), val(Beg))) --> begofstm(scn),
       vn(Tag, ' '), [' '], vn(End, ' '), [' '], vn(Beg, ')'), [')'].

/*** stack manipulation statements ***/
/* stinit statement (initialize stack) */
stackstm(Env, stinit(Stack, Exp)) --> begofstm(stinit),
         stidentifier(Env, Stack, ' '),
         [' '], exp(Env, Exp, ')'), [')'].
/* push statement */
stackstm(Env, push(Stack, Exp)) --> begofstm(push),
         stidentifier(Env, Stack, ' '),
         [' '], exp(Env, Exp, ')'), [')'].
/* pop statement */
stackstm(Env, pop(Stack, Tag)) --> begofstm(pop),
         stidentifier(Env, Stack, ' '),
         [' '], identifier(Env, Tag, ')'), [')'].
/* stack copy statement */
stackstm(Env, stcopy(Stack1, Stack2)) --> begofstm(stcopy),
         stidentifier(Env, Stack1, ' '),
         [' '], [id(to)], [' '],
         stidentifier(Env, Stack2, ')'), [')'].
```

```
/*** queue manipulation statements ***/
/* qinit statement (initialize queue) */
queuestm(Env, qinit(Queue, Exp)) --> begofstm(qinit),
         qidentifier(Env, Queue, ' '),
         [' '], exp(Env, Exp, ')'), [')'].
/* insert value in front of queue */
queuestm(Env, inqfront(Queue, Exp)) --> begofstm(inqfront),
         qidentifier(Env, Queue, ' '),
         [' '], exp(Env, Exp, ')'), [')'].
/* insert value in back of queue */
queuestm(Env, inqback(Queue, Exp)) --> begofstm(inqback),
         qidentifier(Env, Queue, ' '),
         [' '], exp(Env, Exp, ')'), [')'].
/* remove value from front of queue */
queuestm(Env, remqfront(Queue, Tag)) --> begofstm(remqfront),
         qidentifier(Env, Queue, ' '),
         [' '], identifier(Env, Tag, ')'), [')'].
/* remove value from back of queue */
queuestm(Env, remqback(Queue, Tag)) --> begofstm(remqback),
         qidentifier(Env, Queue, ' '),
         [' '], identifier(Env, Tag, ')'), [')'].
/* output the contents of a queue */
queuestm(Env, qty(Queue)) --> begofstm(qty),
         qidentifier(Env, Queue, ')'), [')'].
/* copy from one queue to another */
queuestm(Env, qcopy(Queue1, Queue2)) --> begofstm(qcopy),
         qidentifier(Env, Queue1, ' '),
         [' '], [id(to)], [' '],
         qidentifier(Env, Queue2, ')'), [')'].

/*** stop statement ***/
stopstm(stop) --> [' '], ['('], [id(stop)], [')'].

/*** if its not one of the pre-defined statements above ***/
/*** it must be a constructed line possibly containing ***/
/*** processor functions and parameter transformations ***/
constline(Env, cl(Line)) --> [Toks], {build(Line, Toks, Rest),
                              not isnull(Line), isnull(Rest)}.

/*** determine if we have the beginning of a statement ***/
begofstm(Type) --> [' '], ['('], [id(Type)], [' '].

/*** get a variable name ***/
/* get an array name */
identifier(Env, id(array(Tag, Exp)), Endch) --> vn(Tag, '!'), ['!'],
          exp(Env, Exp, Endch), {declare(array(Tag), val([]), Env)}.
/* get a non-array name */
identifier(Env, id(Tag), Endch) --> vn(Tag, Endch),
                                    {declare(Tag, undef, Env)}.
```

```
/* get a stack name */
stidentifier(Env, id(stack(Tag)), Endch) --> vn(Tag, Endch),
            {declare(stack(Tag), val(undef, []), Env)}.
/* get a queue name */
qidentifier(Env, id(queue(Tag)), Endch) --> vn(Tag, Endch),
           {declare(queue(Tag), val(undef, []), Env)}.

/*** general expression handler (quoted strings, functions, ***/
/*** and arithmetic expressions) ***/
exp(Env, Exp, Endch) --> (quote(Exp, noout);
                          arithexp(Env, Exp, Endch)),
                         ({notequal(Exp, expr(error))};
                          {equal(Exp, expr(error))}, vn(_, Endch)).

/*** process quoted values found in assignment statement ***/
/*** expressions, array indices, and output statement ***/
/*** expressions ***/
quote(val(Val), Type) --> {genquote(Quote)}, [Quote],
                          qv(QL, 0, Type), {list(Val, QL)}.

/*** get quoted strings (generate list of ascii codes) ***/
qv([40 | R], PC, Type) --> ['('], {NPC is PC + 1},
                           qv(R, NPC, Type).
qv([], 0, Type), [Delim] --> delimit(Type, Delim).
qv([41 | R], PC, Type) --> [')'], {PC > 0, NPC is PC - 1},
                           qv(R, NPC, Type).
qv([], 0, Type) --> {genquote(Quote)}, [Quote].
qv(List, PC, Type) --> [id(Word)], {list(Word, L1)},
                       qv(L2, PC, Type),
                       {append(L1, L2, List)}.
qv(List, PC, Type) --> [Any], {not isptpf(Any), list(Any, L1)},
                       qv(L2, PC, Type), {append(L1, L2, List)}.

/*** have we reached the delimiter for the quoted string? ***/
/* output statements are delimited by closing parens and commas */
delimit(out, ')') --> [')'].
delimit(out, ',') --> [','].
/* non-output statement values are delimited by closing parens */
delimit(noout, ')') --> [')'].

/*** expression handler for arithmetic expressions ***/
arithexp(Env, Exp, Endch) --> factor(Env, Lh, Endch),
                              restexp(Env, Lh, Exp1, Endch),
                              /* unquoted numbers are treated as */
                              /* identifiers */
                              ({equal(Exp1, val(Val)), number(Val),
                               Exp = deref(id(Val)); Exp = Exp1});
                              {Exp = expr(error)}.
restexp(Env, Lh, Exp, Endch) --> [CH], {isaddsub(CH)},
                                 factor(Env, Rh, Endch),
```

```
                                {op(CH, Lh, Rh, Subexp)},
                                restexp(Env, Subexp, Exp, Endch).
restexp(Env, Lh, Lh, Endch) --> [].
factor(Env, Exp, Endch) --> primary(Env, Lh, Endch),
                            restfactor(Env, Lh, Exp, Endch).
restfactor(Env, Lh, Exp, Endch) --> [CH], {ismuldiv(CH)},
                                    primary(Env, Rh, Endch),
                                    {op(CH, Lh, Rh, Subexp)},
                                    restfactor(Env, Subexp, Exp, Endch).
restfactor(Env, Lh, Lh, Endch) --> [].
primary(Env, Exp, Endch) --> func(Env, Exp);
                             number(Exp);
                             expid(Env, Exp, Endch);
                             ['('], arithexp(Env, Exp, ')'), [')'].


/*** process functions ***/
/* handles the size function call */
func(Env, size(Exp)) --> ['('], [id(size)], [' '],
                         exp(Env, Exp, ')'), [')'].
/* handles the concat function call */
func(Env, concat(Exp1, Exp2)) --> ['('], [id(concat)], [' '],
                                  exp(Env, Exp1, ' '), [' '],
                                  exp(Env, Exp2, ')'), [')'].
/* handles the top of stack function call */
func(Env, top(Tag)) --> ['('], [id(top)], [' '],
                        stidentifier(Env, Tag, ')'), [')'].
/* handles the stack size function call */
func(Env, stsize(Tag)) --> ['('], [id(stsize)], [' '],
                           stidentifier(Env, Tag, ')'), [')'].
/* handles the stack empty function call */
func(Env, stempty(Tag)) --> ['('], [id(stempty)], [' '],
                            stidentifier(Env, Tag, ')'), [')'].
/* handles the front of queue function call */
func(Env, front(Tag)) --> ['('], [id(front)], [' '],
                          qidentifier(Env, Tag, ')'), [')'].
/* handles the back of queue function call */
func(Env, back(Tag)) --> ['('], [id(back)], [' '],
                         qidentifier(Env, Tag, ')'), [')'].
/* handles the size of queue function call */
func(Env, qsize(Tag)) --> ['('], [id(qsize)], [' '],
                          qidentifier(Env, Tag, ')'), [')'].
/* handles the queue empty function call */
func(Env, qempty(Tag)) --> ['('], [id(qempty)], [' '],
                           qidentifier(Env, Tag, ')'), [')'].


/*** do we have a number ***/
number(val(Val)) --> [id(Val)], {number(Val)}.
```

```
/* check for number preceded by unary minus */
number(val(Val)) --> ['-'], [id(Val1)], {number(Val1), Val is -Val1}.

/*** determine variable name for variables in arithmetic ***/
/*** expressions (they cannot contain arithmetic operators) ***/
/* process variables preceded by unary minus */
expid(Env, times(val(-1), deref(id(Tag))), Endch) --> ['-'],
      expvn(Tag, Endch), {not isnull(Tag), firstch(Tag, CH),
      not number(CH), notequal(CH, '(')}.
expid(Env, deref(id(Tag)), Endch) --> expvn(Tag, Endch),
                                      {not isnull(Tag),
                                       firstch(Tag, CH),
                                       not number(CH),
                                       notequal(CH, '(')}.
expvn('', Endch), [CH] --> [CH], {isaddsub(CH); ismuldiv(CH);
                           equal(CH, ')'); equal(CH, Endch)}.
expvn(Tag, Endch) --> [id(ID)], expvn(Tag1, Endch),
                      {concat(ID, Tag1, Tag)}.
expvn(Tag, Endch) --> [CH], {not isaddsub(CH), not ismuldiv(CH),
                      notequal(CH, ')'), notequal(CH, id(_)),
                      not isptpf(CH), notequal(CH, Endch)},
                      expvn(Tag1, Endch), {concat(CH, Tag1, Tag)}.

/*** determine variable name for an identifier ***/
vn(Tag, Endch) --> ['('], vn(Tag1, ')'), {concat('(', Tag1, Tag2),
                   concat(Tag2, ')', Tag3)}, [')'], vn(Tag4, Endch),
                   {concat(Tag3, Tag4, Tag)}.
vn('', Endch), [Endch] --> [Endch].
vn(Tag, Endch) --> [id(ID)], vn(Tag1, Endch),
                   {concat(ID, Tag1, Tag)}.
vn(Tag, Endch) --> [CH], {notequal(CH, Endch), notequal(CH, ')'),
                   notequal(CH, '('), notequal(CH, id(_)),
                   not isptpf(CH), notequal(CH, ' ')},
                   vn(Tag1, Endch), {concat(CH, Tag1, Tag)}.

/*** expression handler for ty (output) statement ***/
/*** creates a list of values or dereferenced identifiers ***/
/*** for output ***/
outexp(Env, []), [')'] --> [')'].
outexp(Env, List) --> [','], outexp(Env, List).
/* output quoted values */
outexp(Env, [F | R]) --> quote(F, out), outexp(Env, R).
/* output function values */
outexp(Env, [F | R]) --> func(Env, F), outexp(Env, R).
/* output variable values */
outexp(Env, [F | R]) --> (identifier(Env, Tag, ',');
                          identifier(Env, Tag, ')')),
                         {F = deref(Tag)}, outexp(Env, R).
```

```
/*** build a constructed line ***/
build([], [], []).
/* process parameter transformations */
build([pt(Param, Num) | Rest]) --> [[#, Param, Num]],
                                   {integer(Param)},
                                   build(Rest).
/* process processor function 1 */
build([pf(1) | Rest]) --> [[#, F, 14]], {not integer(F)},
                          build(Rest).
/* process other processor functions */
build([pf(Type) | Rest]) --> [[#, F, Type]], {not integer(F)},
                             build(Rest).
/* process non-pt/pf's (regular values) */
build([val(CL) | Rest]) --> clv(CL), {not isnull(CL)}, build(Rest).

/*** build a constructed line until we get to a pt or pf ***/
clv('', [], []).
clv('', [[#, Y, Z] | Rest], [[#, Y, Z] | Rest]).
clv(Val, [id(Word) | R1], R2) :- clv(Val1, R1, R2),
                                 concat(Word, Val1, Val).
clv(Val, [CH | R1], R2) :- atom(CH), clv(Val1, R1, R2),
                           concat(CH, Val1, Val).

                     /*********************************/
                    /*****   semantic portion   *****/
                   /*********************************/

/*** process the list of statements in abstract syntax form ***/
sememe([S1 | S2], state(M, Cont, I, O, C3, L, R), St2) :-
       sememe(S1, state(M, [S2 | Cont], I, O, C3, L, R), St1),
       continuation(St1, St2).
sememe([]) --> [].

/*** process the individual abstract syntax statements ***/
sememe(setq(id(Tag), Exp)) --> sememe(Exp, Val),
                               ({equal(Val, val(error))},
                                update(Tag, val(0));
                               update(Tag, Val)).
sememe(execute(Exp)) --> sememe(Exp, id(Tag)), lookup(Tag, Val),
                         execute(Val).
sememe(input(Exp)) --> sememe(Exp, id(Tag)),
                       (transput(in, Val), update(Tag, Val);
                        iocherror).
sememe(output(List)) --> outval(List, '', Val), transput(out, Val).
sememe(cbk(id(Tag))) --> lookup(Tag, Val), cbk(Val).
sememe(scn(id(Array), id(Tag), Num)) --> (lookup(Tag, Val);
                                          {Val = val(0)}),
                                         (lt(Val, val(2), val(true)),
                                          {End = val(1)};
```

```
                                        {End = Val}),
                                        (eq(Num, val(0), val(true)),
                                         {Index = val(1)};
                                        {Index = val(0)}),
                                        scn(Array, Index, End).
sememe(stinit(id(Stack), Exp)) --> sememe(Exp, Max),
                                   update(Stack, val(Max, [])).
sememe(push(id(Stack), Exp)) --> sememe(Exp, Val), push(Stack, Val).
sememe(pop(id(Stack), id(Tag))) --> pop(Stack, Tag).
sememe(stcopy(id(Stack1), id(Stack2))) -->
       lookup(Stack1, val(Max1, Stvals1)),
       ({equal(Max1, undef)},
         stackerror(Stack1, 'not initialized, stcopy ignored');
        {notequal(Max1, undef)},
         lookup(Stack2, val(Max2, Stvals2)),
         ({equal(Max2, undef)},
           stackerror(Stack2, 'not initialized, stcopy ignored');
          {notequal(Max2, undef), length(Stvals1, 0, Len1)},
           ({gt(val(Len1), Max2, val(true))},
             stackerror(Stack2, 'overflow occurred, stcopy ignored');
            {le(val(Len1), Max2, val(true))},
             update(Stack2, val(Max2, Stvals1))))).
sememe(qinit(id(Queue), Exp)) --> sememe(Exp, Max),
                                  update(Queue, val(Max, [])).
sememe(inqfront(id(Queue), Exp)) --> sememe(Exp, Val),
                                     inq(Queue, Val, front).
sememe(inqback(id(Queue), Exp)) --> sememe(Exp, Val),
                                    inq(Queue, Val, back).
sememe(remqfront(id(Queue), id(Tag))) --> remq(Queue, Tag, front).
sememe(remqback(id(Queue), id(Tag))) --> remq(Queue, Tag, back).
sememe(qty(id(Queue))) --> lookup(Queue, val(Max, Qvals)),
       ({equal(Max, undef)},
         qerror(Queue, 'not initialized, qty ignored');
        qty(Qvals)).
sememe(qcopy(id(Queue1), id(Queue2))) -->
       lookup(Queue1, val(Max1, Qvals1)),
       ({equal(Max1, undef)},
         qerror(Queue1, 'not initialized, qcopy ignored');
        {notequal(Max1, undef)},
         lookup(Queue2, val(Max2, Qvals2)),
         ({equal(Max2, undef)},
           qerror(Queue2, 'not initialized, qcopy ignored');
          {notequal(Max2, undef), length(Qvals1, 0, Len1)},
           ({gt(val(Len1), Max2, val(true))},
             qerror(Queue2, 'overflow occurred, qcopy ignored');
            {le(val(Len1), Max2, val(true))},
             update(Queue2, val(Max2, Qvals1))))).
sememe(stop) --> newstate(continuation, []),
```

```
                    newstate(result, stopped).
sememe(cl(List)) --> cl(List, '', CL), ({isnull(CL)};
                     execute(val(CL))).

/*** process the arithmetic expressions ***/
sememe(plus(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                  sememe(Exp2, Val2),
                                  {add(Val1, Val2, Val)};
                                  exprerror, {Val = val(error)}.
sememe(minus(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                   sememe(Exp2, Val2),
                                   {subtract(Val1, Val2, Val)};
                                   exprerror, {Val = val(error)}.
sememe(times(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                   sememe(Exp2, Val2),
                                   {mult(Val1, Val2, Val)};
                                   exprerror, {Val = val(error)}.
sememe(division(Exp1, Exp2), Val) --> sememe(Exp1, Val1),
                                      sememe(Exp2, Val2),
                                      {divide(Val1, Val2, Val)};
                                      exprerror, {Val = val(error)}.

/*** process functions ***/
sememe(size(Exp), val(Val)) --> sememe(Exp, Val1),
                                {size(Val1, Val)}.
sememe(concat(Exp1, Exp2), val(Val)) --> sememe(Exp1, val(Val1)),
                                         sememe(Exp2, val(Val2)),
                                         {concat(Val1, Val2, Val)}.
/* process stack functions */
sememe(top(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = ''},
         stackerror(Stack, 'not initialized, returning null for top');
        {notequal(Max, undef), firstelem(Stvals, val(Val))}).
sememe(stsize(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = 0},
       stackerror(Stack, 'not initialized, returning zero for stsize');
        {notequal(Max, undef), length(Stvals, 0, Val)}).
sememe(stempty(id(Stack)), val(Val)) -->
       lookup(Stack, val(Max, Stvals)),
       ({equal(Max, undef), Val = true},
      stackerror(Stack, 'not initialized, returning true for stempty');
        {notequal(Max, undef)},
         ({isnull(Stvals), Val = true};
          {not isnull(Stvals), Val = false})).
/* process queue functions */
sememe(front(id(Queue)), val(Val)) -->
       lookup(Queue, val(Max, Qvals)),
```

```
        ({equal(Max, undef), Val = ''},
          qerror(Queue, 'not initialized, returning null for front');
         {notequal(Max, undef), firstelem(Qvals, val(Val))}).
sememe(back(id(Queue)), val(Val)) -->
        lookup(Queue, val(Max, Qvals)),
        ({equal(Max, undef), Val = ''},
          qerror(Queue, 'not initialized, returning null for back');
         {notequal(Max, undef), lastelem(Qvals, val(Val))}).
sememe(qsize(id(Queue)), val(Val)) -->
        lookup(Queue, val(Max, Qvals)),
        ({equal(Max, undef), Val = 0},
          qerror(Queue, 'not initialized, returning zero for qsize');
         {notequal(Max, undef), length(Qvals, 0, Val)}).
sememe(qempty(id(Queue)), val(Val)) -->
        lookup(Queue, val(Max, Qvals)),
        ({equal(Max, undef), Val = true},
          qerror(Queue, 'not initialized, returning true for qempty');
         {notequal(Max, undef)},
          ({isnull(Qvals), Val = true};
           {not isnull(Qvals), Val = false})).

/*** process all other expressions ***/
sememe(deref(Exp), Val) --> sememe(Exp, id(Tag)),
                            (lookup(Tag, Val); {Val = val('')}).
sememe(id(Tag), id(Tag)) --> [].
sememe(val(Val), val(Val)) --> [].
sememe(expr(error), val(0)) --> exprerror.

/*** continue with the next statement on the continuation list ***/
/* normal continuation */
continuation(state(M, [S2 | Cont], I, O, C3, L, R), St2) :-
             (equal(R, ok); equal(R, skipping(O, _))),
             sememe(S2, state(M, Cont, I, O, C3, L, ok), St2).
/* continuation list is empty */
continuation(state(M, [], I, O, C3, L, R),
             state(M, [], I, O, C3, L, R)).
/* skip SkC statements (we are not in a loop) */
continuation(state(M, Cont, I, O, C3, noloop, skipping(SkC, IOC)),
              St2) :-
             SkC > 0, skip(SkC, NewSkC, Cont, NewCont),
             (equal(NewSkC, 0),
              sememe(NewCont, state(M, [], I, O, C3, noloop, ok),
                              St2);
             NewSkC > 0, isnull(NewCont),
             St2 = state(M, [], I, O, C3, noloop,
                         skipping(NewSkC, IOC))).
/* skip SkC statements (we are in a loop, IOC is iteration */
/* open count) */
continuation(state(M, Cont, I, O, C3, loop, skipping(SkC, IOC)),
```

```
            St2) :-
           SkC > 0, skip(SkC, NewSkC, IOC, NewIOC, Cont, NewCont),
           (NewIOC < 1, /* we have skipped out of the loop */
            St2 = state(M, [NewCont], I, O, C3, noloop,
                        skipping(NewSkC, NewIOC));
           NewIOC > 0,
            equal(NewSkC, 0), /* we are still in the loop */
            sememe(NewCont, state(M, [], I, O, C3, loop, ok),
                           St2)).

/*** skip lines (no loops involved) ***/
skip(SkC, SkC, [[]], []).
skip(1, 0, [[St | Rest]], Rest).
skip(SkC, NewSkC, [[St | Rest]], NewCont) :-
     SkC > 1, SkC1 is SkC - 1,
     skip(SkC1, NewSkC, [Rest], NewCont).
skip(SkC, 0, [Cont], Cont) :- SkC < 1.

/*** skip lines from within a loop ***/
skip(SkC, SkC, IOC, IOC, [[]], []).
skip(1, 0, IOC, NewIOC, [[St | Rest]], Rest) :-
     newioc(St, IOC, NewIOC).
skip(SkC, NewSkC, IOC, NewIOC, [[St | Rest]], NewCont) :-
     SkC > 0,
     SkC1 is SkC - 1, newioc(St, IOC, IOC1),
     (equal(IOC1, 0), NewIOC is 0, NewSkC is SkC1, NewCont = Rest;
     IOC1 > 0, skip(SkC1, NewSkC, IOC1, NewIOC, [Rest], NewCont)).
skip(SkC, 0, IOC, IOC, [Cont], Cont) :- SkC > 1.

/*** update iteration open count if necessary ***/
newioc(cl([pf(8) | Rest]), IOC, NewIOC) :- NewIOC is IOC - 1.
newioc(cl([pf(7) | Rest]), IOC, NewIOC) :- NewIOC is IOC + 1.
newioc(St, IOC, IOC) :- notequal(St, cl([pf(8) | Rest])),
                        notequal(St, cl([pf(7) | Rest])).

/*** look up the value of a variable ***/
/* look up the value of an array variable */
lookup(array(Tag, Exp), Val) --> sememe(Exp, Index),
                                 getstate(memory, Mem),
                                 {lookup(array(Tag), Mem, val(List)),
                                 lookupa(Index, List, Val),
                                 notequal(Val, undef)}.
/* look up the value of an ordinary variable */
lookup(Tag, Val) --> getstate(memory, Mem), {lookup(Tag, Mem, Val)}.
lookup(Tag, [loc(Tag, Val) | R], Val) :-
       notequal(Val, undef), not var(Val),
       (equal(Val, val(V)); equal(Val, val(V, _))),
       not var(V).
lookup(Tag, [loc(Tag1, V) | Rest], Val) :- notequal(Tag, Tag1),
```

```
                                                lookup(Tag, Rest, Val).
/* look up the specific value of array variable for the */
/* particular index */
lookupa(Index, [], undef).
lookupa(val(Index), [Index, Val | Rest], Val).
lookupa(Index, [Index1, Val1 | Rest], Val) :-
        lookupa(Index, Rest, Val).

/*** set a new value for a variable ***/
/* set a new value for an array variable */
update(array(Tag, Exp), Val) --> sememe(Exp, Index),
      getstate(memory, Mem1),
      {lookup(array(Tag), Mem1, val(List)),
      updatea(Index, Val, List, Newlist),
      update(array(Tag), val(Newlist), Mem1, Mem2)},
      newstate(memory, Mem2).
/* set a new value for an ordinary variable */
update(Tag, Val) --> getstate(memory, Mem1),
                     {update(Tag, Val, Mem1, Mem2)},
                     newstate(memory, Mem2).
update(Tag, Val, [], [loc(Tag, Val) | _]).
update(Tag, Val, [loc(Tag, V) | Env], [loc(Tag, Val) | Env]).
update(Tag, Val, [L | Env1], [L | Env2]) :-
       equal(L, loc(Tag1, V)), notequal(Tag, Tag1), (var(Env1),
       Env2 = [loc(Tag, Val) | _]; update(Tag, Val, Env1, Env2)).
/* set the specific value of array variable for the */
/* particular index */
updatea(val(Index), Newval, [], [Index, Newval]).
updatea(val(Index), Newval, [Index, Oldval | Rest],
        [Index, Newval | Rest]).
updatea(Index, Newval, [Index1, Val1 | Rest],
        [Index1, Val1 | Newrest]) :-
        notequal(Index, val(Index1)),
        updatea(Index, Newval, Rest, Newrest).

/*** execute a statement by sending it through all three ***/
/*** phases (lexical, syntax, and semantic) ***/
execute(Val, state(Mem, C, I, O, C3, L, R),
             state(Mem1, C1, I1, O1, C3a, L1, R1)) :-
        Val = val(Code), list(Code, Text1),
        /* handle stop specially */
        (equal(Code, ' (stop)'), C1 = [], R1 = stopped,
          Mem1 = Mem, I1 = I, O1 = O;
        C1 = C, append(Text1, [36,10], Text),
        lexemes(Toks, Text, []),
        stmtrain(Mem, Tree, Toks, []), !,
        (isclfunc(Tree),
          /* unrecognized statements output on channel 3 */
         transput(chan3, Val, state(Mem, C, I, O, C3, L, R),
```

```
                                state(Mem1, C1, I1, O1, C3a, L1, R1));
        not isclfunc(Tree),
        sememe(Tree, state(Mem, [], I, O, C3, L, R),
                     state(Mem1, [], I1, O1, C3a, L1, R1)))).

/*** construct an output line for the ty statement ***/
outval([], Val, val(Val)) --> [].
outval([Exp | Rest], Val, NewVal) --> sememe(Exp, val(Val1)),
                                      {concat(Val, Val1, Val2)},
                                      outval(Rest, Val2, NewVal).

/*** produce a new input/output list ***/
transput(in, Val, state(Mem, C, In, O, C3, L, ok),
                  state(Mem, C, In1, O, C3, L, ok)) :-
                  io(Val, In, In1).
transput(out, Val, state(Mem, C, I, Out, C3, L, ok),
                   state(Mem, C, I, Out1, C3, L, ok)) :-
                   io(Val, Out, Out1).
transput(chan3, Val, state(Mem, C, I, O, Chan3, L, ok),
                     state(Mem, C, I, O, Chan3a, L, ok)) :-
                     io(Val, Chan3, Chan3a).

/*** get an input value or add a new output value ***/
io(val(Val)) --> [val(Val)].

/*** process the cbk statement ***/
cbk(val(Val)) --> {index(Val, '[', Front, Len, Back)},
                  checkstub(Val, Front, Len, Err1),
                  ({equal(Err1, error)},
                   update(err, val(1));
                  {index(Back, ']', Entries, _, _)},
                  checkentries(Back, Entries, Err2),
                  ({equal(Err2, error)},
                   update(err, val(1));
                  lookup(fh, val(Vfh)),
                  ({Vfh >= Len};
                  update(fh, val(Len))))).

/*** check the stub of the codebook condition or action ***/
checkstub(Whole, Whole, Len, error) -->
          cbkerror('** error unbalanced or missing brackets').
checkstub(Whole, Stub, 0, error) -->
          cbkerror('** error no stub for condition or action').
checkstub(Whole, Stub, Len, error) --> {Len > 38},
          cbkerror('** error stub length > 38 chars.').
checkstub(Whole, Stub, Len, ok) --> {notequal(Whole, Stub),
          Len > 0, Len =< 38}.

/*** check the entries of the codebook condition or action ***/
checkentries(Whole, Whole, error) -->
```

```
            cbkerror('** error unbalanced or missing brackets').
checkentries(Whole, Entries, Error) -->
            {index(Entries, ',', Entry, Len, Rest)},
            checklen(Len, Err1),
            ({equal(Err1, error), Error = error};
            lookup(bh, val(Val)),
            ({Val >= Len};
            update(bh, val(Len))),
            ({isnull(Rest), Error = ok};
            checkentries(Whole, Rest, Error))).

/*** check the length of an entry ***/
checklen(Len, error) --> {Len > 38},
        cbkerror('** error entry length > 38 chars.').
checklen(Len, ok) --> {Len =< 38}.

/*** process the scn statement ***/
scn(Tag, val(Index), val(End)) --> {Index > End}.
scn(Tag, val(Index), val(End)) --> {Index =< End},
    sememe(output([deref(id(array(Tag, val(Index))))])),
    {NewI is Index + 1}, scn(Tag, val(NewI), val(End)).

/*** push a value onto a stack ***/
push(Stack, Val) --> lookup(Stack, val(Max, Stvals)),
     ({equal(Max, undef)},
       stackerror(Stack, 'not initialized, push ignored');
      {notequal(Max, undef), length(Stvals, 0, Depth)},
       ({equ(Max, val(Depth), val(true))},
         stackerror(Stack, 'stack overflow, push ignored');
        {lt(val(Depth), Max, val(true))},
        update(Stack, val(Max, [Val | Stvals])))).

/*** pop a value onto a stack ***/
pop(Stack, Tag) --> lookup(Stack, val(Max, Stvals)),
    ({equal(Max, undef)},
      stackerror(Stack, 'not initialized, pop ignored');
     {notequal(Max, undef)},
     ({isnull(Stvals)},
       stackerror(Stack, 'pop on empty stack ignored');
      {not isnull(Stvals), Stvals = [Val | Rest]},
       update(Tag, Val), update(Stack, val(Max, Rest)))).

/*** insert a value onto the front or back of a queue ***/
inq(Queue, Val, ForB) --> lookup(Queue, val(Max, Qvals)),
    ({equal(Max, undef)},
      qerror(Queue, 'not initialized, queue insertion ignored');
     {notequal(Max, undef), length(Qvals, 0, Depth)},
      ({equ(Max, val(Depth), val(true))},
        qerror(Queue, 'queue overflow, insertion ignored');
       {lt(val(Depth), Max, val(true))},
     .
```

```
        ({equal(ForB, front)},
          update(Queue, val(Max, [Val | Qvals]));
         {equal(ForB, back), append(Qvals, [Val], NewQvals)},
          update(Queue, val(Max, NewQvals))))).

/*** remove a value from the front or back of a queue ***/
remq(Queue, Tag, ForB) --> lookup(Queue, val(Max, Qvals)),
    ({equal(Max, undef)},
      qerror(Queue, 'not initialized, queue removal ignored');
     {notequal(Max, undef)},
      ({isnull(Qvals)},
        qerror(Queue, 'removal from empty queue ignored');
       {not isnull(Qvals)},
        ({equal(ForB, front), Qvals = [Val | Rest]},
          update(Queue, val(Max, Rest)), update(Tag, Val);
         {equal(ForB, back), lastelem(Qvals, Val)},
          update(Tag, Val), {rmlastelem(Qvals, NewQvals)},
          update(Queue, val(Max, NewQvals))))).

/*** output the contents of a queue ***/
qty([]) --> [].
qty([Val | Rest]) --> transput(out, Val), qty(Rest).

/*** evaluate a constructed line with parameter ***/
/*** transformations (pt) and processor functions (pf) ***/
cl([], CL, CL) --> [].
/* normal values in the constructed line (not a pt or pf) */
cl([val(Val) | Rest], CL, FinalCL) --> {concat(CL, Val, NewCL)},
                                       cl(Rest, NewCL, FinalCL).
/* pt 0 - copy parameter to constructed line */
cl([pt(P,0) | Rest], CL, FinalCL) --> getpval(P, Val, Err),
                                      {concat(CL, Val, NewCL)},
                                      cl(Rest, NewCL, FinalCL).
/* pt 1 - copy variable value to constructed line */
cl([pt(P,1) | Rest], CL, FinalCL) --> getpval(P, Tag, Err),
                                      (lookup(Tag, val(Val));
                                     {Val = ''}),
                                      {concat(CL, Val, NewCL)},
                                      cl(Rest, NewCL, FinalCL).
/* pt 4 - copy parameter treated as arithmetic expression */
/* to constructed line */
cl([pt(P,4) | Rest], CL, FinalCL) --> getpval(P, Exp, Err),
                                      ({equal(Err, error),
                                       concat(CL, '', NewCL)};
                                      {equal(Err, ok)},
                                      expression(Exp, Val1),
                                      ({(equal(Val1, 32768);
                                         equal(Val1, -32768)),
                                        Val = '-/////';
```

```
                                        inrange(Val1, Val)}),
                                       {concat(CL, Val, NewCL)}),
                                       cl(Rest, NewCL, FinalCL).
/* pt 0 - copy parameter length to constructed line */
cl([pt(P,5) | Rest], CL, FinalCL) --> getpval(P, Val, Err),
                                       ({equal(Err, error),
                                        concat(CL, '', NewCL)};
                                       {size(val(Val), Len),
                                       concat(CL, Len, NewCL)}),
                                       cl(Rest, NewCL, FinalCL).
/* pt 6 - reset value of parameter */
cl([pt(P,6) | Rest], CL, FinalCL) --> update(param(P), val(CL)),
                                       cl(Rest, '', FinalCL).
/* pt 7 - context-controlled interation */
cl([pt(P,7) | Rest], '', '') --> [].
cl([pt(P,7) | Rest], CL, '') --> (lookup(param(P), Save);
                                  {Save = undef}),
                                  {breakch(Rest, Break)},
                                  ({equal(Break, error)},
                                   update(param(P), val(''));
                                  getstate(continuation, [Body]),
                                  newstate(continuation, []),
                                  getstate(loopstate, SaveL),
                                  newstate(loopstate, loop),
                                  {listofcl(Break, CL, CLlist)},
                                  dopt7(P, CLlist, Body),
                                  getstate(result, Result),
                                  newstate(result, ok),
                                  update(param(P), Save),
                                  newstate(result, Result),
                                  newstate(loopstate, SaveL)).
/* pf 0 - terminate processing */
cl([pf(0) | Rest], CL, '') --> sememe(stop).
/* pf 1 - output constructed line without rescanning */
cl([pf(1) | Rest], '', FinalCL) --> converror,
                                       cl(Rest, '', FinalCL).
cl([pf(1) | Rest], CL, FinalCL) --> {notequal(CL, '')},
                                       transput(out, val(CL)),
                                       cl(Rest, '', FinalCL).
/* pf 3 - set the value of a variable */
cl([pf(3) | Rest], CL, FinalCL) --> getpval(1, Tag, Err1),
                                       ({equal(Err1, error)};
                                       {equal(Err1, ok)},
                                       getpval(2, Val, Err2),
                                       update(Tag, val(Val))),
                                       cl(Rest, '', FinalCL).
/* pf 4 - set skip counter unconditionally */
cl([pf(4) | Rest], CL, FinalCL) --> getpval(1, Exp, Err),
```

```
                                    expression(Exp, Val),
                                    ({Val =< 0},
                                     cl(Rest, '', FinalCL);
                                    newstate(result, skipping(Val, 1)),
                                    ({isnull(Rest)};
                                     newstate(continuation, []))).
/* pf 5 - set skip counter conditionally on string equality */
cl([pf(50) | Rest], CL, FinalCL) --> dopf5(50, Rest, FinalCL).
cl([pf(51) | Rest], CL, FinalCL) --> dopf5(51, Rest, FinalCL).
/* pf 6 - set skip counter conditionally on relative values */
/* of two arithmetic expressions */
cl([pf('6-') | Rest], CL, FinalCL) --> dopf6('6-', Rest, FinalCL).
cl([pf(60) | Rest], CL, FinalCL) --> dopf6(60, Rest, FinalCL).
cl([pf(61) | Rest], CL, FinalCL) --> dopf6(61, Rest, FinalCL).
cl([pf('6+') | Rest], CL, FinalCL) --> dopf6('6+', Rest, FinalCL).
/* pf 7 - count-controlled iteration */
cl([pf(7) | Rest], CL, FinalCL) --> {isnull(CL)},
                                    cl(Rest, '', FinalCL);
                                    {FinalCL = ''},
                                    getstate(continuation, [Cont]),
                                    ({isnull(Rest), Body = Cont;
                                     append([cl(Rest)], Cont, Body)}),
                                    newstate(continuation, []),
                                    getstate(loopstate, SaveL),
                                    newstate(loopstate, loop),
                                    expression(CL, Val),
                                    ({Val =< 0}, dopf7(Body, 1);
                                     dopf7(Body, Val)),
                                    newstate(loopstate, SaveL).
/* pf 8 - advance an iteration */
cl([pf(8) | Rest], CL, FinalCL) --> getstate(loopstate, noloop),
                                    cl(Rest, '', FinalCL).
cl([pf(8) | Rest], CL, '') --> getstate(loopstate, loop),
                               getstate(continuation, [Cont1]),
                               ({isnull(Rest), Cont2 = Cont1;
                                append([cl(Rest)], Cont1, Cont2)}),
                               {append([cl([pf(8)])], Cont2, Cont3)},
                               {append([[]], Cont3, Cont)},
                               newstate(continuation, Cont).
/* pf 9 - escape from the current macro */
cl([pf(9) | Rest], CL, '') --> newstate(result, escaped),
                               newstate(continuation, []).
/* pf i - set the value of a parameter to the next input value */
cl([pf(I) | Rest], CL, FinalCL) --> ({firstch(I, i)},
   ({not lastch(I, 4)},
   transput(out, val('error -- unexpected call on unknown file')),
   iocherror; ({lastch(CL, Param), integer(Param)},
   (transput(in, Val), update(param(Param), Val); iocherror);
```

```
   converror))),
   cl(Rest, '', FinalCL).

/*** execute the body of context-controlled iteration ***/
dopt7(Param, [], Body) --> rmpf8.
dopt7(Param, [NewP | Rest], Body) --> update(param(Param), NewP),
                                      sememe(Body),
                                      (checkskip;
                                      getstate(continuation, []);
                                      ({isnull(Rest)};
                                       newstate(continuation, [])),
                                      dopt7(Param, Rest, Body)).

/*** perform processor function 5 ***/
dopf5(Type, Rest, FinalCL) --> (getpval(1, Val1, Err1),
                                ({equal(Err1, error)},
                                 cl(Rest, '', FinalCL);
                                getpval(2, Val2, Err2),
                                ({equal(Err2, error)},
                                 cl(Rest, '', FinalCL);
                                ({dopfrel(Type, Val1, Val2)},
                                 cl(Rest, '', FinalCL);
                                getpval(3, Exp, Err3),
                                expression(Exp, Val),
                                ({Val =< 0},
                                 cl(Rest, '', FinalCL);
                                newstate(result, skipping(Val, 1)),
                                ({isnull(Rest), FinalCL = Rest};
                                 newstate(continuation, [])))))))).

/*** perform processor function 6 ***/
dopf6(Type, Rest, FinalCL) --> (getpval(1, Exp1, Err1),
                                ({equal(Err1, error)},
                                 cl(Rest, '', FinalCL);
                                expression(Exp1, Val1),
                                getpval(2, Exp2, Err2),
                                ({equal(Err2, error)},
                                 cl(Rest, '', FinalCL);
                                expression(Exp2, Val2),
                                ({dopfrel(Type, Val1, Val2)},
                                 cl(Rest, '', FinalCL);
                                getpval(3, Exp3, Err3),
                                expression(Exp3, Val),
                                ({Val =< 0},
                                 cl(Rest, '', FinalCL);
                                newstate(result, skipping(Val, 1)),
                                ({isnull(Rest), FinalCL = Rest};
                                 newstate(continuation, [])))))))).
```

```
/*** perform the relationships for pf 5 and pf6 ***/
dopfrel(50, Val1, Val2) :- eq(val(Val1), val(Val2), val(false)).
dopfrel(51, Val1, Val2) :- eq(val(Val1), val(Val2), val(true)).
dopfrel('6-', Val1, Val2) :- lt(val(Val1), val(Val2), val(false)).
dopfrel('6+', Val1, Val2) :- gt(val(Val1), val(Val2), val(false)).
dopfrel(60, Val1, Val2) :- equ(val(Val1), val(Val2), val(false)).
dopfrel(61, Val1, Val2) :- equ(val(Val1), val(Val2), val(true)).

/*** execute the body of a count-controlled iteration ***/
dopf7(Body, 0) --> rmpf8.
dopf7(Body, Count) --> sememe(Body),
                       (checkskip; getstate(continuation, []);
                       {NewC is Count - 1},
                       ({equal(NewC, 0)};
                        newstate(continuation, [])),
                       dopf7(Body, NewC)).

/*** remove parameter trans. 8 from the continuation list ***/
rmpf8 --> getstate(continuation, Cont1),
          {Cont1 = [cl([pf(8)]) | Cont]},
          newstate(continuation, [Cont]).

/*** lookup the value of a parameter ***/
getpval(Param, Val, Err) --> lookup(param(Param), val(Val)),
                             {Err = ok, !};
                             converror, {Val = '', Err = error}.

/*** get the current state of the machine ***/
getstate(memory, M, state(M, C, I, O, C3, L, R),
                    state(M, C, I, O, C3, L, R)).
getstate(continuation, C, state(M, C, I, O, C3, L, R),
                          state(M, C, I, O, C3, L, R)).
getstate(input, I, state(M, C, I, O, C3, L, R),
                   state(M, C, I, O, C3, L, R)).
getstate(output, O, state(M, C, I, O, C3, L, R),
                    state(M, C, I, O, C3, L, R)).
getstate(chan3, C3, state(M, C, I, O, C3, L, R),
                    state(M, C, I, O, C3, L, R)).
getstate(loopstate, L, state(M, C, I, O, C3, L, R),
                       state(M, C, I, O, C3, L, R)).
getstate(result, R, state(M, C, I, O, C3, L, R),
                    state(M, C, I, O, C3, L, R)).

/*** set a new state for the machine ***/
newstate(memory, Val, state(M, C, I, O, C3, L, R),
                      state(Val, C, I, O, C3, L, R)).
newstate(continuation, Val, state(M, C, I, O, C3, L, R),
                            state(M, Val, I, O, C3, L, R)).
newstate(input, Val, state(M, C, I, O, C3, L, R),
                     state(M, C, Val, O, C3, L, R)).
```

```
newstate(output, Val, state(M, C, I, O, C3, L, R),
                      state(M, C, I, Val, C3, L, R)).
newstate(chan3, Val, state(M, C, I, O, C3, L, R),
                     state(M, C, I, O, Val, L, R)).
newstate(loopstate, Val, state(M, C, I, O, C3, L, R),
                         state(M, C, I, O, C3, Val, R)).
newstate(result, Val, state(M, C, I, O, C3, L, R),
                      state(M, C, I, O, C3, L, Val)).

/*** determine valid break characters for pt 7 ***/
breakch([val(Val)], val(Val)).
breakch([], val('')).
breakch(Any, error) :- not isnull(Any), notequal(Any, [val(Val)]).

/*** divide the constructed line into parts based on the break ***/
/*** characters for pt 7 ***/
listofcl(val(''), CL, List) :- breakup(CL, List).
listofcl(val(Break), CL, List) :- rmpar(CL, CL1),
                                  list(Break, BL), list(CL1, CLL),
                                  (bal(CLL, CLL, [], 0),
                                  breakup(BL, CLL, List);
                                  List = [undef, val('')]).

/*** remove the outer parenthesis from the constructed line ***/
rmpar(CL, CL) :- firstch(CL, CH), notequal(CH, '('),
                 lastch(CL, CH1), notequal(CH1, ')').
rmpar(CL, CL2) :- rmlpar(CL, CL1), rmrpar(CL1, CL2).
/* remove the left parenthesis */
rmlpar(CL, CL1) :- list(CL, [40 | R]), bal(R, R, [], 0),
                   list(CL1, R); CL1 = CL.
/* remove the right parenthesis */
rmrpar(CL, CL1) :- list(CL, List), lastelem(List, 41),
                   bal(List, List, [], 1), rmlastelem(List, CLL),
                   list(CL1, CLL); CL1 = CL.

/*** break up the constructed line for pt 7 ***/
/* no break characters - break up constructed line into single */
/* characters */
breakup(CL, List) :- atomic(CL), list(CL, L1), brkup(L1, List).
brkup([], []).
brkup([F | R], [val(F1) | R1]) :- list(F1, [F]), brkup(R, R1).
/* break up constructed line by searching for a break character */
breakup(BL, [], []).
breakup(BL, CLL, [val(F) | R]) :- not isnull(CLL),
                                  search(BL, CLL, Left, F1),
                                  list(F, F1), breakup(BL, Left, R).

/*** search for a break character in the constructed line */
search(BL, [], [], []).
/* do not search inside balanced parenthesis */
```

```
search(BL, [40 | R], NewCL, [40 | NewP]) :- bal(R, NewPL, CL, 0),
                                            search(BL, CL, NewCL, R1),
                                            append(NewPL, R1, NewP).
search(BL, [F | R], R, []) :- match(BL, F).
search(BL, [F | R], NewCL, [F | R1]) :- not match(BL, F),
                                        search(BL, R, NewCL, R1).

/*** try to match a single character from the constructed line ***/
/*** with the break characters ***/
match([F | R], F).
match([F | R], CH) :- match(R, CH).

/*** find the end of balanced parenthesis ***/
bal([40 | R], [40 | R1], CL, PC) :- NPC is PC + 1, bal(R, R1, CL, NPC).
bal([], [], [], 0).
bal([41 | R], [41], R, 0).
bal([41 | R], [41 | R1], CL, PC) :- PC > 0, NPC is PC - 1,
                                    bal(R, R1, CL, NPC).
bal([Any | R], [Any | R1], CL, PC) :- notequal(Any, 40),
                                      notequal(Any, 41),
                                      bal(R, R1, CL, PC).

/*** evaluate an unformed (not in abstract syntax form) ***/
/*** arithmetic expression by sending it through lexemes, ***/
/*** arithexp of morpheme, and sememe ***/
expression('', 0) --> [].
expression(Exp, Val) --> {list(Exp, Elist1),
                          append(Elist1, [41, 36, 10], Elist),
                          lexemes([Toks], Elist, [])},
                          exphand(Toks, Exp1),
                          sememe(Exp1, val(Val1)),
                          ((({isnull(Val1)}; {equal(Val1, error)};
                             {not number(Val1)}, exprerror),
                             {Val = 0});
                           {Val = Val1}).
/* convert the expression to abstract syntax */
exphand(Tokens, Exp, state(Mem, C, I, O, C3, L, R),
                     state(Mem, C, I, O, C3, L, R)) :-
       arithexp(Mem, Exp1, ')', Tokens, Rest),
       ((equal(Rest, [')']); equal(Rest, [')',')'])),
       getehv(Exp1, Exp);
       Exp = expr(error)).

/*** get a proper expression for the expression handler ***/
/* arithexp treats unquoted numbers as identifiers but exphand */
/* doesn't like that */
getehv(deref(id(Val)), val(Val)) :- number(Val), !.
getehv(Exp, Exp). /* all other expressions are ok */

/*** see if a stop statement has been executed ***/
/*** (#f0, #f9, or (stop)) ***/
```

```
checkstop --> getstate(result, stopped).
checkstop --> getstate(result, escaped).
/*** see if skipping is taking place ***/
checkskip --> getstate(result, skipping(Num, IOC)).

/*** generate an error message ***/
/* expression error */
exprerror --> transput(out, val('********* expr error')).
/* conversion error */
converror --> transput(out, val('********* conv error')).
/* input/output channel error (fatal) */
iocherror --> transput(out, val('********* ioch error')),
              newstate(continuation, []),
              newstate(result, 'I/O error').
/* output an error message for the codebook statement */
cbkerror(Errmess) --> transput(out, val(Errmess)).
/* output a stack error message */
stackerror(stack(Stack), Mess) -->
          sememe(output([val('** error on stack '),val(Stack),
                         val(': '),val(Mess)])).
/* output a queue error message */
qerror(queue(Queue), Mess) -->
       sememe(output([val('** error on queue '),val(Queue),
                      val(': '),val(Mess)])).

/*** perform the actual arithmetic operations ***/
add(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                     Z1 is X1 + Y1, inrange(Z1, Z).
subtract(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                          Z1 is X1 - Y1, inrange(Z1, Z).
mult(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                      Z1 is X1 * Y1, inrange(Z1, Z).
divide(X, Y, val(Z)) :- checkval(X, X1), checkval(Y, Y1),
                        Z1 is X1 // Y1, inrange(Z1, Z).
/* make sure a number is in the proper range */
/* (if not then truncate) */
inrange(Int, Int) :- value(n1, Max), Int =< Max, Int >= -Max, !.
inrange(Num, Int) :- value(n2, Bits1), value(n3, Bits2),
                     Int is (Num<<(Bits2-Bits1))>>(Bits2-Bits1).

/*** perform the actual relational operations ***/
eq(X, Y, val(true)) :- X == Y.
eq(X, Y, val(false)) :- not X == Y.
equ(X, Y, val(true)) :- checkval(X, X1), checkval(Y, Y1), X1 =:= Y1.
equ(X, Y, val(false)) :- checkval(X, X1), checkval(Y, Y1),
                         not X1 =:= Y1.
lt(X, Y, val(true)) :- checkval(X, X1), checkval(Y, Y1),
                       X1 < Y1.
lt(X, Y, val(false)) :- checkval(X, X1), checkval(Y, Y1),
```

```
                          X1 >= Y1.
gt(X, Y, val(true)) :- checkval(X, X1), checkval(Y, Y1),
                         X1 > Y1.
gt(X, Y, val(false)) :- checkval(X, X1), checkval(Y, Y1),
                          X1 =< Y1.
le(X, Y, val(true)) :- gt(X, Y, val(false)).
le(X, Y, val(false)) :- gt(X, Y, val(true)).
ge(X, Y, val(true)) :- lt(X, Y, val(false)).
ge(X, Y, val(false)) :- lt(X, Y, val(true)).

/*** make sure a value is really a number ***/
checkval(val(''), 0).
checkval(val(Val), Val) :- number(Val).

/*** perform the size function ***/
size(val(Val1), Val) :- list(Val1, L), length(L, 0, Val).

                   /************************************/
                  /*****   syntactic constraints   *****/
                 /************************************/

/*** declare adds a declaration to the environment if ***/
/*** the variable is not already a member ***/
declare(Tag, Val, Env) :- member(loc(Tag, _), Env).
declare(Tag, Val, Env) :- not member(loc(Tag, _), Env),
                          addword(loc(Tag, Val), Env).

/*** member handles lists with uninstantiated tail ***/
member(X, [Y]) :- var(Y), !, fail.
member(X, [X | Y]).
member(X, [Y | Z]) :- notequal(X, Y), member(X, Z).

/*** add a word to the end of a list ***/
addword(Label, [X | Y]) :- var(X), var(Y), X = Label.
addword(Label, [X | Rest]) :- not var(X),
                              addword(Label, Rest).

/*** append the 2nd list to the end of the 1st list ***/
/*** giving the 3rd list ***/
append([U | V], W, [U | X]) :- append(V, W, X).
append([], X, X).

/*** return the first element of a list ***/
firstelem([Head | Tail], Head).
firstelem([], '').

/*** find the last element of a list ***/
lastelem([Last], Last).
lastelem([X | Y], Last) :- lastelem(Y, Last).

/*** remove the last element of a list ***/
rmlastelem([Elem], []).
rmlastelem([F | R], [F | R1]) :- rmlastelem(R, R1).
```

```
/*** find the length of a list ***/
length([], N, N).
length([Head | Rest], Sofar, Total) :- More is Sofar + 1,
                                       length(Rest, More, Total).

/*** generate a single quote (apostrophe) character ***/
genquote(Quote) :- list(Quote, [39]).

/*** concatenate two atoms (Val1 and Val2) to form a new ***/
/*** atom (NewVal) ***/
concat('', Val2, Val2) :- !.
concat(Val1, '', Val1) :- !.
/* plus signs require special handling so we don't lose them */
concat('+', Val2, NewVal) :- integer(Val2),
                             NewVal =.. ['+', Val2], !.
concat(Val1, Val2, NewVal) :- isplus(Val1, Arg),
                              not isplus(Val2, _),
                              concat(Arg, Val2, Val3),
                              NewVal =.. ['+', Val3], !.
concat(Val1, Val2, NewVal) :- not isplus(Val1, _),
                              isplus(Val2, Arg),
                              concat(Val1, '+', Val3),
                              concat(Val3, Arg, NewVal), !.
concat(Val1, Val2, NewVal) :- isplus(Val1, Arg1),
                              isplus(Val2, Arg2),
                              concat(Arg1, '+', Val3),
                              concat(Val3, Arg2, Val4),
                              NewVal =.. ['+', Val4], !.
concat(Val1, Val2, NewVal) :- list(Val1, L1), list(Val2, L2),
                              append(L1, L2, L3),
                              list(NewVal, L3), !.

/*** extract the first character of an atom ***/
firstch(Atom, Ch) :- not isnull(Atom), list(Atom, [First | Rest]),
                     list(Ch, [First]).

/*** extract the last character of an atom ***/
lastch(Atom, Ch) :- not isnull(Atom), list(Atom, List),
                    lastelem(List, L), list(Ch, [L]).

/*** find the single character (Char) in the string (Str) ***/
/*** and return the front of the string (Front), the length ***/
/*** of the front (Len), and the back of the string (Back) ***/
/*** (minus the character) ***/
index(Str, Char, Front, Len, Back) :- list(Str, StrL),
      list(Char, ChL), indexl(StrL, ChL, FrL, BkL, Len, 0),
      list(Front, FrL), list(Back, BkL).
indexl([CH | Rest], [CH], [], Rest, Count, Count).
indexl([], [CH], [], [], Count, Count).
indexl([First | Rest], [CH], [First | Front], Back, Total, Sofar) :-
```

```
        notequal(CH, First), NewC is Sofar + 1,
        indexl(Rest, [CH], Front, Back, Total, NewC).

/*** convert an atom to a list of ascii codes and ***/
/*** vice versa (same as built-in functor name but ***/
/*** works with null arguments) ***/
list('', []) :- !.
list(Atom, List) :- name(Atom, List).

isnull([]).              /* is arg. the null list */
isnull('').              /* is arg. the null string */
      /* is arg. a parameter trans. or proc. function */
isptpf(['#', P, N]).
      /* is arg. the cl functor (constructed line) */
isclfunc([cl(_)]).
     /* does Term contain an addition operation */
isplus(Term, Arg) :- Term =.. ['+', Arg].
     /* is arg. the ascii code for a newline char. */
isnewline(10).
     /* is arg. the ascii code for a end-of-line char. */
iseol(36).
    /* is arg. one of the ascii codes for an alphanumeric char. */
ischar(CH) :- CH >= 65, CH =< 90; CH >= 97, CH =< 122;
              CH >= 48, CH =< 57.
isaddsub('+').           /* is arg. an addition sign */
isaddsub('-').           /* is arg. an subtraction sign */
ismuldiv('*').           /* is arg. an multiplication sign */
ismuldiv('/').           /* is arg. an division sign */

/*** convert an arithmetic operation to abstract notation ***/
op('+', Lh, Rh, plus(Lh, Rh)).
op('-', Lh, Rh, minus(Lh, Rh)).
op('*', Lh, Rh, times(Lh, Rh)).
op('/', Lh, Rh, division(Lh, Rh)).

/*** instantiate the args. to be the same thing ***/
/*** (fail if they are already instantiated ***/
/*** to different things) ***/
notequal(X, Y) :- not(equal(X, Y)).
equal(X, Y) :- X = Y.

/*** various machine dependent values ***/
value(n1, 32767).          /* value of largest integer in ASP */
value(n2, 16).             /* bits per integer for ASP */
value(n3, 32).             /* bits per integer for this prolog */
```

APPENDIX 2.8

FEATURES AND FOLLIES OF THE CODE BODY FORMAL DEFINITION

Listed here are several strong points about the definition along with explanations of why they are worth special attention. Then several weak points about the definition, brought out by thorough examination and testing, are expounded upon.

Strong points about the Code Body definition:

o use of continuations for goto statement

The development of continuations by Strachey and Wadsworth was an important advance in the descriptive techniques of semantics. It led to simpler and smoother descriptions of various constructs, some of which would be impossible to describe without continuations [Gordon, 1979; Strachey & Wadsworth, 1974]. We have adapted the method of "impure continuations" to relational semantics in order to describe the goto statement, modeled after the work of Moss [1981]. The work was made somewhat more difficult than the Barrel–F definition by the fact that goto's can branch into or out of loops.

o additional output channel (channel 3 as well as 4)

Two output files are include in the definition in contrast to the one in the Barrel–F definition. This demonstrates the feasibility of defining the many input and output files available in the actual ASP implementation.

o promotes concept of a constructed line and its submission for possible execution

ASP provides the concept of the constructed line in code bodies where lines are built up by evaluating any parameter transformations and processor functions in the line. The resulting text is treated as a call to another definition. If none of the

definition's templates are matched the line is output to channel 3. The definition supports the constructed line concept.

o error messages match those of ASP

Errors are handled by a separate "status" parameter in the state of the machine. When an error occurs in the execution of a program the error message that is generated is the same as that generated by the ASP processor.

o execute statement sends the value of the variable through the 3 phases of the definition (lexical, syntactic, and semantic)

Many Barrel–F statements can be executed from within code bodies. The execute statement is one of those. It causes the value of a variable to be executed as if it were a Barrel–F statement. The semantic definition of the execute statement actually sends the value of the variable through the 3 phases of the definition as if it were a one statement program.

o implementation details can be included

Syntactic constraints which are machine dependent can be specified in the definition such as a limit on the size of integers. Checks can also be performed in the semantics to assure such constraints are followed at run time.

Problems with the Code Body definition:

o statements are not limited to one line (80 characters)

The definition does not complain about statements that extend beyond the current line. The ASP implementation, however, does not allow statements to span more than one line. Nor does it allow more than 80 characters in a single line.

o very large strings of digits are converted to floating point numbers upon input

In ASP numbers are treated as character strings until an arithmetic operation is performed on them. Thus, very big numbers are allowed. But the definition (because of the way Prolog treats numbers) converts very long strings of digits to

floating point numbers upon input. A similar problem occurs with leading zeros in numbers. The Prolog processor strips leading zeros upon input whereas the ASP processor allows them to be part of the number until an arithmetic operation is performed.

o the "target escape character", zero, space, bracket, and arithmetic operation symbols cannot be specified

The ASP processor allows the user to change the symbol used to mark parameter transformations, and processor functions. In addition, the user can change the symbol used to indicate a zero, space, brackets, and the four arithmetic operations. The definition assumes a specific character will be used all the time for each of these symbols.

APPENDIX 2.9

PROGRAMS USED TO TEST THE CODE BODY DEFINITION

The programs used to test the formal definition of ASP code bodies are listed below.

Test 1 – Test of processor function 3.

```
#f3$
$
```

Test 2 – Test of processor function 3.

```
val#26 #f3$
$
```

Test 3 – Test of processor function 3.

```
var#16 #f3$
...#11...#f14$
$
```

Test 4 – Test of processor function 3.

```
var#16 #f3$
$
```

Test 5 – Test of processor function 3.

```
var#16 val#26 #f3$
...#11...#f14$
$
```

Test 6 – Test of processor function 4.

```
#f4 hm#f14$
bad#f14$
good#f14$
$
```

Test 7 – Test of processor function 4.

```
2n#16 #f4 hm#f14$
bad#f14$
```

```
good#f14$
$
```

Test 8 – Test of processor function 4.

```
2#16 #f4$
bad1$
bad2$
good1$
good2$
$
```

Test 9 – Test of processor function 4.

```
4#f7$
3#16 #f4$
4#f7$
bad1$
#f8$
ok1$
#f8$
$
```

Test 10 – Test of processor function 4.

```
4#f7$
4#16 #f4$
#f7$
4#f7$
bad1$
#f8$
ok1$
#f8$
$
```

Test 11 – Test of processor function 4.

```
4#f7$
2#16 #f4$
4#f7$
bad1$
ok1$
#f8$
ok2$
#f8$
$
```

Test 12 – Test of processor function 4.

```
4#f7$
ok1$
```

```
1#16 #f4$
bad1#f8$
ok2$
#f8$
bye$
$
```

Test 13 – Test of processor function 4.

```
4#f7$
2#16 #f4$
#f7$
#f8$
loop2$
#f8$
$
```

Test 14 – Test of processor function 5.

```
a#16 b#26 n*2#36 #f51 hm#f14$
bad#f14$
good#f14$
$
```

Test 15 – Test of processor function 5.

```
#16 b#26 1#36 #f51$
bad#f14$
good#f14$
$
```

Test 16 – Test of processor function 5.

```
#16 1#36 #f50$
bad#f14$
good#f14$
$
```

Test 17 – Test of processor function 5.

```
a#16 a#26 2n#36 #f50 hm#f14$
bad#f14$
good#f14$
$
```

Test 18 – Test of processor function 5.

```
a#16 a#26 2n#36 #f51 hm#f14$
bad#f14$
good#f14$
$
```

Test 19 – Test of processor function 5.

```
a#16 a#26 #f50 hm#f14$
bad#f14$
good#f14$
$
```

Test 20 – Test of processor function 6.

```
#16 1#36 #f6+ hm#f14$
bad#f14$
good#f14$
$
```

Test 21 – Test of processor function 6.

```
1#16 2n#26 1#36 #f6+$
bad#f14$
good#f14$
$
```

Test 22 – Test of processor function 6.

```
1#16 2n#26 1#36 #f6-$
bad#f14$
good#f14$
$
```

Test 23 – Test of processor function 6.

```
a#16 a#26 #f3$
a#26 2#36 #f60$
false#f14$
#f9$
true#f14$
$
```

Test 24 – Test of processor function 6.

```
2#16 1#26 #f6+ hm#f14$
bad#f14$
good#f14$
$
```

Test 25 – Test of processor function 6.

```
1#16 2#26 #f6+ hm#f14$
bad#f14$
good#f14$
$
```

Test 26 – Test of processor function 6.

```
2#16 1#26 2n#36 #f6- hm#f14$
bad#f14$
good#f14$
$
```

Test 27 – Test of processor function 6.

```
2#16 1#26 2n#36 #f6+ hm#f14$
bad#f14$
good#f14$
$
```

Test 28 – Test of processor function 6.

```
1#16 2#26 n*2#36 #f6- hm#f14$
bad#f14$
good#f14$
$
```

Test 29 – Test of processor function 6.

```
#16 1#26 1#36 #f6-$
bad#f14$
good#f14$
$
```

Test 30 – Test of processor function 6.

```
1#16 1#36 #f6+$
bad#f14$
good#f14$
$
```

Test 31 – Test of processor function 6.

```
2n#16 1#26 1#36 #f6-$
bad#f14$
good#f14$
$
```

Test 32 – Test of the setq, incr, decr, bump, and zero statements.

```
(setq a '1)$    <-- this symbol ($) is necessary for a comment
(incr a)
(ty a)
(decr a)
(ty a)
(bump a 2*a+3)
```

```
 (ty a)
 (setq b a)
 (ty b)
 (zero a)
 (ty a)
 (stop)
$
```

Test 33 – A program to find the factorial of n.

```
 (ty 'enter a positive integer)$  this program computes factorial
 (readch '4' n)$                  using a mixture of pre-defined
 (setq i '1)$                     high-level instructions and
 (setq fact '1)$                  machine instructions
n#16 #11#16 0#26 5#36 #f60$
1000#f7$
i#16 #11#16 n#26 #21#26 3#36 #f60$
 (incr i)$
 (setq fact fact*i)$
#f8$
 (ty 'the factorial of ,n,' is ,fact)$
$
```

Test 34 – A program to find the factorial of n.

```
enter a positive integer#f14$  this program computes factorial
2#fi4$                         using machine instructions (no
n#16 #f3$                      pre-defined high-level instructions)
i#16 1#26 #f3$
fact#16 #f3$
n#16 #11#16 0#26 5#36 #f60$
1000#f7$
i#16 #11#16 n#26 #21#26 3#36 #f60$
i#16 i+1#26 #24#26 #f3$
fact#16 fact*i#26 #24#26 #f3$
#f8$
n#16 fact#26$
the factorial of #11 is #21#f14$
$
```

Test 35 – A general test of many different processor functions and parameter transformations.

```
 there#16$
hello#10$
 (setq a 'be)$
a#26 you #21 there$
(2+2*3)/4#96$
```

```
#94$
1,(2;3),4#27;,$
...#20...$
#20#47$
..#40..$
first#f8 second#f8 goodbye$
hello#f14$
var#16 val#26$
hello #f3$
#11#f14$
5#fi4$
...#50...#f14$
(3-1)*2#f7$
*#f14$
#f8$
2n#f7$
*#f14$
#f8$
2-1#16 #f4$
bad news#f14$
a#16 a#26 1#36 #f50$
bad news#f14$
2-1#16 0+1#26 1#36 #f60$
bad news#f14$
#10 is #15 chars. long#f14$
#f9$  #f0 works the same way
bad news#f14$
$
```

Test 36 – Test of combination of parameter transformations and Barrel–F statements.

```
hello#16$
 (setq a '#10)$
 (ty a)$
a#16$
 (setq b #10)
 (ty b)
 (ty #10)
x#16 'hello#26$
 (setq #10 #20)
 (ty #10)
 (zero #10)
 (ty x)
1#26$
 (bump #10 '#20)
 (ty x)
```

```
 (incr #10)
 (ty x)
 (decr #10)
 (ty x)
 (readch '4' #10)
 (ty x)
 (execute #10)
stop#16$
 (#10)
$
```

Test 37 – Test of the execute statement.

```
 (setq a '1)$
 (ty a)
 (readch '4' code)$  input should be (setq a '2)
 (ty code)
 (execute code)
 (ty a)
$
```

Test 38 – Test of the execute statement.

```
 (setq a '1)$
 (ty a)
 (readch '4' code)$  input should be (setq b '2)
 (ty code)
 (execute code)
 (readch '4' code)$  input should be (ty b)
 (ty code)
 (execute code)
$
```

Test 39 – Test of the ty statement.

```
 (setq s 'yucky)$
 (setq t 'pooh)
 (ty 's is ,s,' t is ,t)
 (ty s,'is s)
 (ty '(a,b))
 (ty 'hello there)
 (ty '(a b))
 (ty 's = ,s)
 (setq a '1,&%2)
 (ty 'a is ,a)
$
```

Test 40 – Test of the size function.

```
 (setq n 'abc)$
 (setq a (size n))
 (ty a)
 (setq a (size 'abcd))
 (ty a)
 (setq b '1)
 (setq c '12)
 (setq a (size b+c))
 (ty a)
 (setq a (size b))
 (ty a)
 (ty 'hello,(size c*c),a)
 (setq a 2*(size 'abc)+1)
 (ty a)
 (ty (size (size 'abcdefghij)))
$
```

Test 41 – Test of the plink operator.

```
 (setq arr!'0' 'hello)$
 (setq c '1)
 (setq arr!c c+1)
 (readch '4' arr!c+1)
 (setq t0 arr!'0')
 (setq t1 arr!'1')
t0#16 #11#16 t1#26 #21#26 2#36 #f50
 (ty 'good news)
1#16 #f4
 (ty 'bad news)
 (zero b!'10')
100#f7
 (ty arr!b!'10')
 (setq t b!'10')
t#16 #11#16 2#26 2#36 #f60
 (incr b!'10')
#f8
 (setq arr!'## one' 'any value)
 (ty arr!'## one)
 (setq b '## one)
 (ty arr!b)
$
```

Test 42 – Test of the concat function.

```
 (setq a (concat '1st half' '2nd half))$
 (ty a)
 (setq a (concat '1st half' '2nd half'))
 (ty a)
 (setq b '2nd half)
```

```
 (setq a (concat '1st half' b))
 (ty a)
 (setq b '1st half)
 (setq a (concat b '2nd half))
 (ty a)
 (ty (concat 1+3*2 (concat '7' (size '1234567))))
$
```

Test 43 – Test of the cbk statement.

```
 (setq tab!'0' 't < 0 [])$
 (setq tab!'1' 'make is [cord,reo,dues])
 (setq tab!'2' 'condition is good,bad])
 (setq tab!'3' 'comm is [1%,5%)
 (setq tab!'4' '[needed,not needed])
 (setq tab!'5' 'this stub is going to be much much too long [1,2])
 (setq tab!'6' 'stub [this entry is going to be much too long,2])
 (setq tab!'7' 'managers ok is [needed, not needed])
 (zero fh)
 (zero bh)
 (cbk tab!'0')
 (ty fh,bh)
 (cbk tab!'1')
 (ty fh,bh)
 (cbk tab!'2')
 (ty fh,bh)
 (cbk tab!'3')
 (ty fh,bh)
 (cbk tab!'4')
 (ty fh,bh)
 (cbk tab!'5')
 (ty fh,bh)
 (cbk tab!'6')
 (ty fh,bh)
 (cbk tab!'7')
 (ty fh,bh)
$
```

Test 44 – Test of the scn statement.

```
 (setq cond!'0' 'rem3)$
 (setq cond!'1' 'make is [cord,reo,dues])
 (setq cond!'2' 'cond is [good,bad])
 (setq cond!'3' 'comm is [1%,5%])
 (setq cond!'4' 'shopwork is [needed,not needed])
 (setq c '4)
 (scn cond c 0)
 (scn cond c 1)
$
```

Test 45 – Test of parameter transformation 4.

```
n#16 3#26 #f3$
2+-n#16$
...#14...#f14$
$
```

Test 46 – Test of parameter transformation 4.

```
2n#16$
...#14...$
$
```

Test 47 – Test of qinit, inqfront, and remqfront statements.

```
 (qinit q '2)$
 (inqfront q 'frog)
 (remqfront q var)
 (ty var)
$
```

Test 48 – Test of qcopy statement.

```
 (qinit q1 '3)$
 (qinit q2 '2)
 (inqfront q1 'lilly)
 (inqback q1 'pad)
 (qcopy q1 to q2)
 (remqfront q2 var)
 (ty var)
 (remqfront q2 var)
 (ty var)
$
```

Test 49 – Test of the front of queue function.

```
 (qinit q '2)$
 (inqfront q 'lilly)
 (ty (front q))
 (inqback q 'pad)
 (setq frog (front q))
 (ty frog)
 (remqfront q frog)
 (ty frog,(front q))
$
```

Test 50 – Test of the back of queue function.

```
 (qinit q '2)$
 (inqfront q 'lilly)
```

```
 (ty (back q))
 (inqback q 'pad)
 (setq frog (back q))
 (ty frog)
 (remqfront q frog)
 (ty frog,(back q))
$
```

Test 51 – Test of the queue size function.

```
 (qinit q '2)$
 (ty (qsize q))
 (inqfront q 'lilly)
 (ty (qsize q))
 (inqback q 'pad)
 (setq frog (qsize q))
 (ty frog)
 (remqfront q frog)
 (setq frog (qsize q)+1)
frog#16 #11#16 2#26 2#36 #f60$
 (ty 'whoops)
1#16 #f4$
 (ty 'ok1)
$
```

Test 52 – Test of the queue empty function.

```
 (qinit q '2)$
 (setq emp (qempty q))
emp#96 true#26 2#36$
#91#16 #f50$
 (ty 'whoops1)
1#16 #f4$
 (ty 'ok1)
 (inqfront q 'lilly)
 (setq emp (qempty q))
#91#16 false#26 #f50$
 (ty 'whoops2)
1#16 #f4$
 (ty 'ok2)
 (inqback q 'pad)
 (setq emp (qempty q))
#91#16 true#26 #f50$
 (ty 'ok3)
1#16 #f4$
 (ty 'whoops3)
 (qinit q '3)
 (setq emp (qempty q))
#91#16 #f50$
```

```
 (ty 'whoops4)
1#16 #f4$
 (ty 'ok4)
$
```

Test 53 – Test of the queue empty function.

```
 (qinit q '2)$
 (ty (qempty q))
 (inqfront q 'lilly)
 (setq frog 1+(qempty q))
 (ty frog)
 (inqback q (qempty q))
 (remqback q frog)
 (ty frog)
$
```

Test 54 – Test of the qinit, inqback, remqback statements.

```
 (qinit q '2)$
 (inqback q 'frog)
 (remqback q var)
 (ty var)
$
```

Test 55 – Test of the inqfront and remqfront statements.

```
 (inqfront q 'frog)$    this fails
 (remqfront q wart)
$
```

Test 56 – Test of the inqback and remqback statements.

```
 (inqback q 'frog)$    this fails
 (remqback q wart)
$
```

Test 57 – Test of the overflow and underflow of the inqfront and remqfront statements.

```
 (qinit q '2)$
 (setq var 'frog)
 (inqfront q var)
 (inqfront q (concat 'lilly' 'pad))
 (inqfront q 5*5)
 (remqfront q var1)$
 (ty var1)
 (remqfront q var1)
 (ty var1)
```

```
 (remqfront q var1)
 (ty var1)
$
```

Test 58 – Test of the overflow and underflow of the inqback and remqback statements.

```
 (qinit q '2)$
 (setq var 'frog)
 (inqback q var)
 (inqback q (concat 'lilly' 'pad))
 (inqback q 5*5)
 (remqback q var1)$
 (ty var1)
 (remqback q var1)
 (ty var1)
 (remqback q var1)
 (ty var1)
$
```

Test 59 – Test of the overflow and underflow of the inqfront and remqback statements.

```
 (qinit q '2)$
 (setq var 'frog)
 (inqfront q var)
 (inqfront q (concat 'lilly' 'pad))
 (inqfront q 5*5)
 (remqback q var1)
 (ty var1)
 (remqback q var1)
 (ty var1)
 (remqback q var1)
 (ty var1)
$
```

Test 60 – Test of the overflow and underflow of the inqback and remqfront statements.

```
 (qinit q '2)$
 (setq var 'frog)
 (inqback q var)
 (inqback q (concat 'lilly' 'pad))
 (inqback q 5*5)
 (remqfront q var1)
 (ty var1)
 (remqfront q var1)
```

```
 (ty var1)
 (remqfront q var1)
 (ty var1)
$
```

Test 61 – Test of the qty statement.

```
 (qinit q '3)$
 (setq var 'frog)
 (inqback q var)
 (inqback q (concat 'lilly' 'pad))
 (inqback q 5*5)
 (qty q)
 (remqback q var1)
 (qty q)
 (remqback q var1)
 (qty q)
 (remqback q var1)
 (qty q)
$
```

Test 62 – Test of the stinit, push, and pop statements.

```
 (stinit st '2)$
 (push st 'frog)
 (pop st var)
 (ty var)
$
```

Test 63 – Test of the push and pop statements.

```
 (push st 'frog)$       this fails
 (pop st wart)
$
```

Test 64 – Test of the overflow and underflow of the push and pop statements.

```
 (stinit st '2)$
 (setq var 'frog)
 (push st var)
 (push st (concat 'lilly' 'pad))
 (push st 5*5)
 (pop st var1)$
 (ty var1)
 (pop st var1)
 (ty var1)
 (pop st var1)
 (ty var1)
$
```

Test 65 – Test of the stcopy statement.

```
 (stinit st1 '3)
 (stinit st2 '2)
 (push st1 'lilly)
 (push st1 'pad)
 (stcopy st1 to st2)
 (pop st2 var)
 (ty var)
 (pop st2 var)
 (ty var)
$
```

Test 66 – Test of the top of stack function.

```
 (stinit st '2)
 (push st 'lilly)
 (ty (top st))
 (push st 'pad)
 (setq frog (top st))
 (ty frog)
 (pop st frog)
 (ty frog,(top st))
$
```

Test 67 – Test of the stack size function.

```
 (stinit st '2)
 (ty (stsize st))
 (push st 'lilly)
 (ty (stsize st))
 (push st 'pad)
 (setq frog (stsize st))
 (ty frog)
 (pop st frog)
 (setq frog (stsize st)+1)
frog#16 #11#16 2#26 2#36 #f60$
 (ty 'whoops)
#f9$
 (ty 'ok1)
$
```

Test 68 – Test of the stack empty function.

```
 (stinit st '2)
 (setq emp (stempty st))
emp#96 true#26 2#36$
#91#16 #f50$
 (ty 'whoops1)
```

```
1#16 #f4$
 (ty 'ok1)
 (push st 'lilly)
 (setq emp (stempty st))
#91#16 false#26 #f50$
 (ty 'whoops2)
1#16 #f4$
 (ty 'ok2)
 (push st 'pad)
 (setq emp (stempty st))
#91#16 true#26 #f50$
 (ty 'ok3)
1#16 #f4$
 (ty 'whoops3)
 (stinit st '3)
 (setq emp (stempty st))
#91#16 #f50$
 (ty 'whoops4)
1#16 #f4$
 (ty 'ok4)
$
```

Test 69 – Test of the stack empty function.

```
 (stinit st '2)
 (ty (stempty st))
 (push st 'lilly)
 (setq frog 1+(stempty st))
 (ty frog)
 (push st (stempty st))
 (pop st frog)
 (ty frog)
$
```

APPENDIX 3.1

ASP IMPLEMENTATION OF RUNNABLE SPECIFICATIONS

The following Prolog code serves as the specifications of a decision table presentation processor. The specifications are runnable because Prolog can be executed on a computer. Following the Prolog specifications is the ASP implementation.

```
/* Prolog Specification */

dt :- write('car make ? '), read(C1),
      write('condition ? '), read(C2),
      intermed(C1, C2).

intermed(no, C2) :- write('so long now'), nl.
intermed(C1, C2) :- dec(C1, C2), nl,
                    write('we continue'), nl,
                    dt.

dec(C1, C2) :- table(C1, C2, A1, A2, A3),
               write('commission is '), write(A1), nl,
               write('shop work needed is '), write(A2), nl,
               write('manager ok is '), write(A3), nl.

table(cord, good, A1, A2, A3) :- A1 = '5%',
                                 A2 = 'no-need',
                                 A3 = 'no-req'.
table(cord, poor, A1, A2, A3) :- A1 = '1%',
                                 A2 = '3-weeks',
                                 A3 = 'no-req'.
table(reo, good, A1, A2, A3) :- A1 = '10%',
                                A2 = 'no-need',
                                A3 = 'no-req'.
table(reo, poor, A1, A2, A3) :- A1 = '5%',
                                A2 = '3-weeks',
                                A3 = 'no-req'.
table(duesenberg, good, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.
table(duesenberg, poor, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.
```

```
/* ASP Implementation */

 dt :-
 write('car make ? ') $
 read(C1) $
 write('condition ? ') $
 read(C2) $
 intermedprime(C1, C2) $
$
 intermed(no, #) :-
 write('so long now') $
$
 intermed(#, #) :-
 dec(#10, #20) $
 write('we continue') $
 dt $
$
 dec(#, #) :-
 table(#10, #20, A1, A2, A3) $
 write('commission is ') $
 write(A1) $
 write('shop work is ') $
 write(A2) $
 write('manager ok is ') $
 write(A3) $
$
 table(cord, good, #, #, #) :-
 #10 = '5%' $
 #20 = 'none' $
 #30 = 'not needed' $
$
 table(cord, poor, #, #, #) :-
 #10 = '1%' $
 #20 = '3 weeks' $
 #30 = 'not needed' $
$
 table(reo, good, #, #, #) :-
 #10 = '10%' $
 #20 = 'none' $
 #30 = 'not needed' $
$
 table(reo, poor, #, #, #) :-
 #10 = '5%' $
 #20 = '3 weeks' $
 #30 = 'not needed' $
$
 table(duesenberg, good, #, #, #) :-
 #10 = 'variable' $
```

```
 #20 = '6 weeks' $
 #30 = 'needed' $
$
 table(duesenberg, poor, #, #, #) :-
 #10 = 'variable' $
 #20 = '6 weeks' $
 #30 = 'needed' $
$
:  Support macros written using Barrel's BBAS kit.
$
 intermedprime(#, #) :-
 intermed(#11, #21) $
$
 # = '#' :-
 (fsetq #10 '#20)$
$
 write('#') :-
 (fty '#10)$
$
 write(#) :-
 (fty #10)$
$
 read(#) :-
 (readch '4' #10)$
$
```

APPENDIX 3.2

ASP IMPLEMENTATION OF RUNNABLE SPECIFICATIONS WITH MULTIPLE VALUES

The following Prolog code serves as the specifications of a decision table presentation processor. The specifications are runnable because Prolog can be executed on a computer. Following the Prolog specifications is the ASP implementation. The implementation allows for using variable names (prefaced with an asterisk) to get multiple values from the table as can be done with the specifications.

```
/* Prolog Specification */

dt :- write('car make ? '), read(C1),
      write('condition ? '), read(C2),
      intermed(C1, C2).

intermed(no, C2) :- write('so long now'), nl.
intermed(C1, C2) :- dec(C1, C2), nl,
                    write('we continue'), nl,
                    dt.

dec(C1, C2) :- table(C1, C2, A1, A2, A3),
               write('commission is '), write(A1), nl,
               write('shop work needed is '), write(A2), nl,
               write('manager ok is '), write(A3), nl.

table(cord, good, A1, A2, A3) :- A1 = '5%',
                                 A2 = 'no-need',
                                 A3 = 'no-req'.
table(cord, poor, A1, A2, A3) :- A1 = '1%',
                                 A2 = '3-weeks',
                                 A3 = 'no-req'.
table(reo, good, A1, A2, A3) :- A1 = '10%',
                                A2 = 'no-need',
                                A3 = 'no-req'.
table(reo, poor, A1, A2, A3) :- A1 = '5%',
                                A2 = '3-weeks',
```

```
                                    A3 = 'no-req'.
table(duesenberg, good, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.
table(duesenberg, poor, A1, A2, A3) :- A1 = 'variable',
                                       A2 = '6-weeks',
                                       A3 = 'req'.

/* ASP Implementation */

 dt :-
 write('car make ? ') $
 read(C1) $
 write('condition ? ') $
 read(C2) $
 intermedprime(C1, C2) $
$
 intermed(no, #) :-
 write('so long now') $
$
 intermed(#, #) :-
 dec(#10, #20) $
 write('we continue') $
 dt $
$
 dec(#, #) :-
 tableprime(#10, #20, A1, A2, A3) $
$
 tableprime(#, #, A1, A2, A3) :-
 table(#10, #20, A1, A2, A3) $
 writetable(A1, A2, A3) $
$
 tableprime(*#, #, A1, A2, A3) :-
 table(cord, #20, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(reo, #20, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(duesenberg, #20, A1, A2, A3) $
 writetable(A1, A2, A3) $
$
 tableprime(#, *#, A1, A2, A3) :-
 table(#10, good, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(#10, poor, A1, A2, A3) $
 writetable(A1, A2, A3) $
$
 tableprime(*#, *#, A1, A2, A3) :-
 table(cord, good, A1, A2, A3) $
 writetable(A1, A2, A3) $
```

```
 table(cord, poor, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(reo, good, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(reo, poor, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(duesenberg, good, A1, A2, A3) $
 writetable(A1, A2, A3) $
 table(duesenberg, poor, A1, A2, A3) $
 writetable(A1, A2, A3) $
$
 writetable(A1, A2, A3) :-
 write('commission is ') $
 write(A1) $
 write('shop work is ') $
 write(A2) $
 write('manager ok is ') $
 write(A3) $
$
 table(cord, good, #, #, #) :-
 #10 = '5%' $
 #20 = 'none' $
 #30 = 'not needed' $
$
 table(cord, poor, #, #, #) :-
 #10 = '1%' $
 #20 = '3 weeks' $
 #30 = 'not needed' $
$
 table(reo, good, #, #, #) :-
 #10 = '10%' $
 #20 = 'none' $
 #30 = 'not needed' $
$
 table(reo, poor, #, #, #) :-
 #10 = '5%' $
 #20 = '3 weeks' $
 #30 = 'not needed' $
$
 table(duesenberg, good, #, #, #) :-
 #10 = 'variable' $
 #20 = '6 weeks' $
 #30 = 'needed' $
$
 table(duesenberg, poor, #, #, #) :-
 #10 = 'variable' $
 #20 = '6 weeks' $
 #30 = 'needed' $
$
```

APPENDIX 3.3

ASP IMPLEMENTATION OF I/O INDIFFERENCE

Following is the ASP implementation of a decision table interpreter which provides I/O indifference (i.e., you can enter values for the actions as well as the conditions).

```
DT|    invoke the pattern builder (can only be called once)
ty Welcome to DT Land$
rem     set up the beginnings of the macro to test against
set pat!0 to tab$                beginning of template
set pat!1 to car make:          $   beginning of line 1 of code body
set pat!2 to condition:          $   line 2
set pat!3 to commission:         $
set pat!4 to shop work needed: $
set pat!5 to managers ok:        $   line 5
rem     set up prompts for each line
set prompt!1 to car make: cord, reo, duesenburg, or ?          $
set prompt!2 to condition: good, poor, or ?                    $
set prompt!3 to commission: 1%, 5%, 10%, variable, or ?        $
set prompt!4 to shop work: not needed, 3 weeks, 6 weeks, or ? $
set prompt!5 to managers ok: required, not required, or ?      $
rem     generate the rest of the template and code body
DI 1 1$
rem     put end of line marker on template
pat!0#16$
set pat!0 to #11|$
rem     set up end of macro marker
set pat!6 to #$#$$
rem     write out the macro
pat_out 6$
$
pat_out #|      write out variables pat!0 thru pat!#10 to file pat
 (write 'pat' pat!'0')$
set counter to 1$
counter#26$
#10#f7$
 (write 'pat' pat!'#21')$
setx counter to counter+1$
#f8$
$
DI # #|         generate the template and code body
pat!0#96$
```

```
ans#56$
 (fty prompt!'#10')$
 (readch '4' ans)$
if ans eq '?' skip 4$
set pat!0 to #91,#51$
pat!#10#66$
set pat!#10 to #61 #51##f14#$$
skip 4$
set pat!0 to #91,##$
pat!#10#66$
set pat!#10 to #61 ###200##f14#$$
#20+1#26$
if #10 = 5 skip 2$
#10+1#16$
DI #14 #24$
$
 go|    how we execute the table
pat_in$         read in the pattern that we have created
T$              compare it with our table
$
pat_in| read in the pattern
 (rewind 'pat)$   rewind the file
 (addmacs 'pat)$  read in the pattern (as a macro)
$
T|      the rules of the decision table
tab,cord,good,5%,not needed,not required$
tab,cord,poor,1%,3 weeks,not required$
tab,reo,good,10%,not needed,not required$
tab,reo,poor,5%,3 weeks,not required$
tab,duesenberg,good,variable,6 weeks,required$
tab,duesenberg,poor,variable,6 weeks,required$
$
tab,#|  to catch the ones that don't match
$
```

APPENDIX 6.1

NEW PROCESSOR FUNCTIONS OF ASP

The processor functions which were added to Stage2 in the development of the ASP processor are described here. The descriptions are modeled after those provided for the original processor functions of Stage2 [Waite, 1973]. Each specification gives the format of a call and the action of that call. In the format specification, digits and upper–case letters denote themselves. Lower–case letters denote classes of characters, as follows:

- d Any digit between 0 and 9, inclusive.
- e Target escape character (fourth character of the flag line).
- m Any digit between 0 and 9, inclusive, or any character between a and z, inclusive.

Each description is accompanied by at least one example which uses the function.

Add Definitions

Format: meFA

Action: Processing of the code body is temporarily halted and macro definitions are read from channel m. They are placed into ASP's internal memory along with the macros read in when ASP was started up. Macros are read until a macro terminated by two target end–of–line flags is encountered. At that point, the element immediately following meFA is ignored, and scanning of the code body resumes with the next element. The macros that were read in can subsequently be called.

Example

```
Macro:  add  a  macro.
        #FA$
        $
```

Input: add a macro.
new macro.
this is the new macro#F1$
$$
new macro.

Output: this is the new macro

Close A Channel

Format: meFC

Action: Channel m and the file associated with it is closed. The channel can subsequently be used with another file. This function is only valid with channel 0 and channels 6 through 35. The element immediately following meFC is ignored, and scanning of the code body resumes with the next element.

Example

Macro: close channel '.
'10#FC$
channel '10 is closed#F1$
$

Input: close channel z.

Output: channel z is closed

Utilize a Graphics Terminal

Format: deFG

Action: The digit d, which must be 0, 1, 2, or 3, determines the action to be taken. If d is 0 then the Gigi graphics terminal, which is assumed to be channel 4, is put into graphics mode. If d is 1, then parameter 1, which is assumed to be a Regis graphics command, is written to channel 4. If d is 2, then the Gigi graphics terminal is taken out of graphics mode. If d is 3, then the graphics attributes of the Gigi graphics terminal are reset to default values. The element immediately following deFG is ignored, and scanning of the code body resumes with the next element.

Example

Macro: (write gigi ').
1#FG$
$

Input: (write gigi p[32,32])

Output: (a point is drawn on the graphics terminal at row 32, column 32)

Input a Line

Format: deFIm

Action: A line is read from channel m and stored as the value of parameter d. If m is omitted, then channel 4 is used. The element immediately following deFIm is ignored, and scanning of the code body resumes with the next element.

Example

```
Macro: read '
       #10#F14$
       2#FI$
       #20#F14$
       $

Input: read type in something please
         (from channel 1)
       this is my input
         (from channel 4)

Output: type in something please
        this is my input
```

Execute a CLI Command

Format: eFK

Action: Temporarily suspend processing of ASP and execute a single command of the operating system's command line interpreter. The command is the value of parameter 1. When the command is finished, the element immediately following eFK is ignored, and scanning of the code body resumes with the next element.

Example

```
Macro: cli '.
       #FK$
       here we are back again#F14$
       $
```

Input: cli date.

Output: Sat Nov 5 13:20:52 CST 1988
here we are back again

Escape to the Operating System

Format: eFL

Action: Temporarily suspend processing of ASP and execute the operating system's command line interpreter. When the command line interpreter is terminated, the element immediately following eFL is ignored, and scanning of the code body resumes with the next element.

Example

Macro: escape.
#FL$
here we are back again#F14$
$

Input: escape.
(CLI commands)

Output: (output from CLI commands)
here we are back again

Execute the Barrel/ASP Editor

Format: eFM

Action: Temporarily suspend processing of ASP and execute another ASP process with the macros for the Barrel/ASP editor, BEDIT, loaded. When the editor is terminated, the element immediately following eFM is ignored, and scanning of the code body resumes with the next element.

Example

Macro: call bedit.
#FM$
here we are back again#F14$
$

Input: call bedit.
(editor commands)

Output: (output from editor commands)
here we are back again

Trace Macro Calls

Format: meFT

Action: Turn tracing of all macros calls on or off. When m is a 1, tracing is turned on. When m is a 0, tracing is turned off. The element immediately following meFT is ignored, and scanning of the code body resumes with the next element.

Example

Macros: trace on.
1#FT$
tracing is on#F14$
$
trace off.
0#FT$
tracing is off#F14$
$

Input: trace on.
trace off.

Output: tracing is on
*** trace *** trace off.
tracing is off

APPENDIX 7.1

EXTENSIONS TO FLUB FOR THE ASP IMPLEMENTATION

Following is a listing of the statements added to the FLUB abstract machine language as defined by William Waite [1973]. These extensions were necessary to implement the ASP processor.

1. message trace to '

   Output a tracing message to the specified channel.

2. call cli1

   Execute a single operating system command.

3. call cli

   Execute the operating systems command line interpreter in order to execute multiple operating system commands.

4. call barreled

   Execute the Barrel text editor.

5. getch ' in w

   Associates a channel number with a file name.

6. stochanp6

   Stores the channel number returned by "getch ' in w" in parameter number 6.

7. stochanp8

   Stores the channel number returned by "getch ' in w" in parameter number 8.

8. close next '

Closes the specified channel.

9. gigi

Provides an interface to a GIGI/Regis graphics terminal by calling the gigi subroutine.

APPENDIX 7.2

FLUB VERSION OF THE ASP PROCESSOR

Following is a listing of the FLUB version of the ASP processor. Lines which have been added or modified from the FLUB version of STAGE2 have comments. The commented FLUB version of STAGE2 may be found in [Waite, 1973].

```
flg i = 0.
val i = 1 + 0.
ptr i = 0 + 0.
read next i.
to 98 if flg i ne 0.
val a = char.
ptr a = 8 + 0.
sto a = i.
flg b = 2.
val b = char.
val c = char.
ptr c = 9 + 0.
val d = char.
val e = char.
ptr e = val e.
val f = char.
ptr f = a + 7.
sto f = 0.
val g = 0 + 0.
ptr h = 5 * 7.
flg j = 1.
ptr j = 0 + 0.
flg l = 1.
val l = 0 - 1.
ptr l = 0 + 0.
val m = char.
ptr m = 0 + 0.
flg n = 0.
val n = char.
flg o = 0.
val o = char.
val p = char.
val q = char.
val r = char.
```

```
ptr r = 0 + 0.
ptr 4 = 7 + 7.
ptr 8 = f + 7.
loc pre01.                entry point for adding definitions
to 01 by d.
loc 01.
get i = a.
read next i.
to 98 if flg i ne 0.
ptr i = c + 0.
val y = 0 + 0.
ptr y = c + 0.
to 02 if ptr m = 0.
ptr m = m - 1.
to 01.
loc 02.
ptr 9 = i + 0.
val i = char.
ptr i = 9 - 7.
to 97 if ptr 8 ge i.
sto 9 = i.
to 04 if val i = 1.
to 03 if val i = a.
val y = y + 1.
to 02 if val i ne b.
ptr b = i + 0.
sto 9 = b.
to 02.
loc 03.
ptr 9 = i + 0.
val i = char.
ptr i = 9 - 7.
sto 9 = i.
to 97 if ptr 8 ge i.
to 03 if val i ne 1.
loc 04.
ptr u = 9 - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
```

```
ptr u = u - 7.
sto u = 3.
ptr u = u - 7.
sto u = 3.
ptr v = u - 7.
sto v = 3.
ptr u = v - 7.
ptr 9 = u + 0.
to 97 if ptr 8 ge 9.
get w = a.
get x = y.
flg y = 0.
ptr z = a + 0.
to 58 by b.
to 50 if flg b = 2.
to 56 if flg y = 0.
to ov10 if flg p = 0.       is tracing turned on?
to tr by g.                 yes, call tracing routine
loc ov10.
ptr g = u + 7.
get w = g.
flg 4 = val 4.
to noalt if flg w ne 4.
ptr u = w + 0.
loc noalt.
sto 9 = 1.
ptr 9 = 9 - h.
sto 9 = j.
ptr j = 9 + h.
ptr 9 = 9 - 7.
sto 9 = c.
ptr 9 = 9 - 7.
sto 9 = d.
ptr 9 = 9 - 7.
sto 9 = k.
ptr k = u + 0.
ptr 9 = 9 - 7.
sto 9 = r.
ptr r = 0 + 0.
ptr c = 9 - 7.
to 97 if ptr 8 ge c.
to 05 by d.
loc 05.
ptr 9 = c + 0.
ptr y = 0 + 0.
loc 06.
to 07 if ptr m = 0.
ptr z = k + 7.
```

```
get k = k.
get i = k.
to 08 if val i = 1.
ptr m = m - 1.
get z = z.
to 06 if flg z ne 3.
ptr y = y + 1.
to 06 if val z = 7.
ptr y = y - 1.
to 06 if val z ne 8.
ptr y = y - 1.
to 06 if ptr y ge 0.
to 06 if ptr r = 0.
ptr u = r - 7.
get y = u.
to 49 if flg y ne 1.
ptr c = r + 0.
get r = r.
to 05.
loc 07.
ptr k = k + 7.
get i = k.
to 09 if flg i = 2.
to 22 if flg i = 3.
ptr i = 9 - 7.
sto 9 = i.
ptr 9 = i + 0.
to 97 if ptr 8 ge 9.
to 07 if flg i = 0.
ptr y = c - 9.
ptr y = y / 7.
ptr y = y - 1.
val y = ptr y.
ptr y = c + 0.
to 04 if val i ne 1.
loc 08.
ptr 9 = j - h.
get j = 9.
ptr 9 = 9 - 7.
get c = 9.
ptr 9 = 9 - 7.
get d = 9.
ptr 9 = 9 - 7.
get k = 9.
ptr 9 = 9 - 7.
get r = 9.
return by d.
loc 09.
```

```
ptr v = j + i.
to 21 if val i = 6.
get y = v.
to 45 if val i = 7.
to 23 if flg y = 3.
get x = y.
to 11 if val i = 0.
to 10 if val i = 1.
to 12 if val i = 2.
to 15 if val i = 4.
ptr x = y + 0.
to 20 if val i = 3.
ptr n = val y.
to 18 if val i = 5.
to 23 if val y ne 1.
ptr n = val x.
to 18 if val i = 8.
message conv to 4.
to 94 by b.
to 07.
loc 10.
ptr v = 9 + 7.
get w = f.
ptr z = f + 0.
to 58 by b.
to 07 if flg y ne 1.
flg i = 0.
get x = y.
loc 11.
to 07 if val y = 0.
get i = x.
ptr x = 9 - 7.
sto 9 = x.
ptr 9 = x + 0.
val y = y - 1.
to 07 if val y = 0.
get x = i.
ptr i = 9 - 7.
sto 9 = i.
ptr 9 = i + 0.
to 97 if ptr 8 ge 9.
val y = y - 1.
to 11.
loc 12.
flg f = 2.              set flag for not adding macro definition
flg b = 2.
get w = f.
ptr z = f + 0.
```

```
to 58 by b.
flg f = 0.              reset flag
flg b = 0.
get x = y.
to 11 if flg y = 1.
ptr y = 8 + 0.
flg y = 1.
ptr 1 = 1 + 1.
ptr x = 1 + 0.
ptr w = 9 + 7.
val y = 0 + 0.
loc 13.
ptr v = x / 5.
ptr z = v * 5.
ptr x = x - z.
val x = ptr x.
ptr x = v + 0.
ptr w = w - 7.
sto w = x.
val y = y + 1.
to 97 if ptr 8 ge w.
to 13 if ptr x ne 0.
loc 14.
get x = w.
ptr w = w + 7.
val x = x + e.
ptr x = 8 + 7.
sto 8 = x.
ptr 8 = x + 0.
to 14 if ptr 9 ge w.
sto 8 = 0.
ptr 8 = 8 + 7.
to 97 if ptr 8 ge 9.
sto u = y.
get x = y.
flg i = 0.
to 11.
loc 15.
to 74 by p.
to 18 if ptr n ge 0.
ptr o = 9 - 7.
to 97 if ptr 8 ge o.
sto 9 = o.
ptr 9 = o + 0.
ptr n = 0 - n.
to 18.
loc 16.
get y = v.
```

```
to 17 if flg y = 1.
ptr v = v - 7.
to 16 if val y ne i.
ptr n = y + 0.
to 18.
loc 17.
ptr y = v + h.
to 23 if ptr y = j.
ptr l = l + 1.
ptr i = l + 0.
sto v = i.
ptr v = v - 7.
get y = v.
flg y = 1.
sto v = y.
ptr n = l + 0.
loc 18.
ptr y = n / 5.
ptr z = y * 5.
ptr x = n - z.
flg x = 0.
val x = ptr x.
ptr n = y + 0.
val g = g + 1.
ptr 8 = 8 + 7.
sto 8 = x.
to 18 if ptr n ne 0.
loc 19.
get x = 8.
ptr 8 = 8 - 7.
val g = g - 1.
val x = x + e.
ptr x = 9 - 7.
sto 9 = x.
ptr 9 = x + 0.
to 19 if val g ne 0.
to 07.
loc 20.
get x = x.
val y = y - 1.
to 20 if val y ne 1.
to 07 if flg x = 1.
ptr x = 9 - 7.
to 97 if ptr 8 ge x.
sto 9 = x.
ptr 9 = x + 0.
to 07.
loc 21.
```

```
sto 9 = 1.
ptr k = k + 7.
ptr y = c - 9.
ptr y = y / 7.
flg y = 0.
val y = ptr y.
ptr y = c + 0.
sto v = y.
ptr c = 9 - 7.
to 05.
loc 22.
ptr v = j + 0.
to 16 if ptr i = 0.
to 08 if val i = 9.
ptr v = v + 7.
ptr k = k + 7.
to 32 if val i = 1.
to 32 if val i = 2.
to 33 if val i = 3.
to 42 if val i = 4.
to 36 if val i = 5.
to 39 if val i = 6.
to 43 if val i = 7.
to 47 if val i = 8.
val w = 5 * 9.
val w = w + 4.
to adfun if val i = w.
val w = w + 2.
to close if val i = w.
val w = w + 4.
to gigi if val i = w.
val w = w + 2.
to infun if val i = w.
val w = w + 2.
to cli1 if val i = w.
val w = w + 1.
to cli if val i = w.
val w = w + 1.
to bed if val i = w.
val w = w + 7.
to trace if val i = w.
to 23 if val i ne 0.
stop.
loc 23.
message conv to 4.
to 94 by b.
to 07.
loc trace.    trace function added by john barrett
```

```
prt x = c - 9.          get length of constructed line
to 23 if prt x ne 7.    can only be one character
get w = c.              get that character
val w = w - e.          convert to an integer
to unt if val w = 0.    turning trace on or off?
flg p = 1.              turn trace on
to 05.
loc unt.
flg p = 0.              turn trace off
to 05.
loc gigi.      gigi graphics function added by john barrett
prt x = c - 9.    length of constructed line
to 23 if prt x ne 7.    can only be one character
get w = c.              get the character (the operation code)
val w = w - e.          convert to an integer
to ov9 if val w ne 1.   if 1 put gigi command in line buffer
get y = v.              get pointer to parameter 1 (the gigi command)
to 23 if flg y = 3.     error if parameter is undefined
get x = y.              get first character of parameter
val z = y + 0.          save length of parameter
loc lp2.
char = val x.           put parameter in line buffer
get x = x.              get next character of parameter
val z = z - 1.
to lp2 if val z ne 0.   any more characters?
char = val 1.           close out the line buffer
loc ov9.
gigi.                   send gigi command to terminal
to 98 if flg w ne 0.    get out unless all is ok
to 05.
loc close.               close function by john barrett
ptr x = c - 9.           length of constructed line
to 23 if ptr x ne 7.     can only be one character
get w = c.               get the character
val y = 6 * 7.
to ov5 if val w ne y.    are we using the asterisk extension?
val x = 9.               yes, get the channel number
to gtch by g.            from parameter 9
ptr x = val 8.           put the file name in
to setp by g.            parameter 8
loc ov5.
val w = w - e.           convert it to an integer
close next w.            close the channel
to 98 if flg w ne 0.     get out if not ok
to 05.
loc setp.                subroutine to set a parameter to a channel #
val x = ptr x.           are we storing in para. 6 or 8?
to ov3 if val x = 8.
```

```
stochanp6.              returns address of channel # in ptr z
to ov4.
loc ov3.
stochanp8.              returns address of channel # in ptr z
loc ov4.
ptr x = x * 7.
ptr y = j + x.          set parameter list pointer
sto y = z.              set parameter
return by g.
loc bed.                barrel editor function by john barrett
call barreled.          call up the barrel line editor
to 05.
loc cli1.               cli1 function by john barrett
get x = v.              get parameter 1 pointer
to 23 if flg x = 3.     error if parameter is undefined
to 23 if val x = 0.     error if parameter is null
val y = x.
loc lp3.                put parameter 1 in line buffer
get x = x.
char = val x.
val y = y - 1.
to lp3 if val y ne 0.
char = val 1.           close out line buffer
call cli1.              call operating system
to 05.
loc cli.                cli function by john barrett
call cli.               call up the command line interpreter
to 05.
loc infun.              input function by john barrett
get x = k.              check for alternate input unit
val w = 4 + 0.          default is channel 4
to defau if flg x = 1.  if no channel  # take the default
to nodef if flg x ne 2. check for parameter trans.
to 23 if val x ne 0.    allow only 0 transformation
ptr v = j + 0.          set parameter pointer
ptr v = v + x.
get x = v.              get pointer to parameter
get x = x.              get parameter
loc nodef.
ptr k = k + 7.          advance code body pointer
val w = x + 0.
val y = 6 * 7.          check for special case asterisk
to ov8 if val w ne y.   signalling a file name in a para.
val x = 9.              parameter 9 holds the file name
to gtch by g.           subroutine to get the channel #
ptr x = val 8.          parameter 8 will hold channel #
to setp by g.           subroutine to set para. to channel #
loc ov8.
```

```
val w = w - e.          convert to an integer
loc defau.
read next w.            get the input
to 98 if flg w ne 0.    get out unless all is ok
ptr 9 = 9 + 7.          point to parameter # to hold input
get z = 9.              get parameter # to hold input
val z = z - e.          convert to an integer
ptr z = val z.
to 23 if ptr z ge 5.    only accept parameters 1-9
to 23 if ptr 0 ge z.
ptr v = j + 0.
ptr z = z * 7.
ptr v = v + z.          set parameter pointer
ptr w = c + 0.          point to pseudo-input space
flg x = 0.
loc lp1.
ptr x = w - 7.
to 97 if ptr 8 ge x.    get out if memory is full
val x = char.
sto w = x.
ptr w = w - 7.
to lp1 if val x ne 1.
ptr w = w + 7.
sto w = 1.              close the input line
ptr w = c - w.          calculate length of input
ptr w = w / 7.
val w = ptr w.
flg w = 0.
ptr w = c + 0.          set pointer to start of input line
sto v = w.              set parameter store
ptr c = ptr x.          next empty space for new line
to 05.
loc gtch.               subroutine to get a channel # from a
ptr x = val x.          file name in a parameter specified
ptr x = x * 7.          by val x
ptr z = j + x.
get x = z.              get parameter pointer
to 23 if flg x = 3.     error if parameter is undefined.
to 23 if val x = 0.     error if parameter is null.
ptr z = x.
val y = x.
loc ilp1.               put file name in the line buffer
get z = z.
char = val z.
val y = y - 1.
to ilp1 if val y ne 0.
char = val 1.           close out the line buffer
getch x in w.           get channel number
```

```
to 05 if flg x ne 0.     ignore it if not ok?
return by g.
loc adfun.               add a macro definition from a macro call
flg b = 2.               set definition flag
flg e = 2.               set add flag
get i = a.               get current input unit
val h = i + 0.           save it
to ov11 if ptr 9 = c.    default is current input unit
ptr g = c - 9.           find length of constructed line
ptr g = g / 7.
to 23 if ptr g ge 2.     error if more than one character
get w = c.               else get new input unit
val y = 6 * 7.           test for asterisk extension
to ov6 if val w ne y.
val x = 9.               get channel number
to gtch by g.
ptr x = val 8.           set parameter 8 to channel number
to setp by g.
loc ov6.
val w = w - e.           make it an integer
val i = w + 0.
sto a = i.               put it away for definition phase
loc ov11.
ptr o = d.               save return address in d
to pre01.                go get the definitions
loc 32.
get x = k.
val w = 3 + 0.
to 24 if flg x = 1.
to ov7 if flg x ne 2.    check for parameter transformation
to 23 if val x ne 0.     allow only zero transformation
ptr z = j + x.           point to parameter pointer
get x = z.               get parameter pointer
get x = x.               get parameter value
loc ov7.
ptr k = k + 7.
val w = x + 0.
val y = 6 * 7.           check for asterisk
to ov1 if val y ne w.
val x = 9.               parameter 9 holds the file name
to gtch by g.            get channel # from file name
ptr x = val 8.           parameter 8 will hold the channel #
to setp by g.            set para. 8 to channel #
loc ov1.
val w = w - e.           convert to an integer
get x = k.
to 24 if flg x = 1.
rewind w.
```

```
ptr k = k + 7.
loc 24.
to 31 if val i = 2.
sto 9 = 1.
ptr x = c + 0.
to 57 if ptr c ne 9.
ptr k = k + 7.
get i = k.
to 25 if flg i ne 1.
ptr k = k - 7.
to 23.
loc 25.
ptr z = val i.
ptr z = z - e.
to 28 if ptr z ge 5.
to 28 if ptr 0 ge z.
val x = i + 0.
ptr z = z * 7.
ptr y = j + z.
get y = y.
to 27 if flg y = 3.
get z = y.
loc 26.
to 27 if val y = 0.
char = val z.
get z = z.
val y = y - 1.
ptr k = k + 7.
get i = k.
to 26 if val i = x.
to 25.
loc 27.
char = val f.
ptr k = k + 7.
get i = k.
to 27 if val i = x.
to 25.
loc 28.
to 57 if flg i = 1.
char = val i.
ptr k = k + 7.
get i = k.
to 25.
loc 31.
get i = a.
to 29 if ptr c = 9.
get x = c.
val y = 6 * 7.          check for asterisk
```

```
to ov2 if val y ne x.
ptr w = x.              save ptr x
val v = w.              save val w
val x = 7.              parameter 7 holds the file name
to gtch by g.           get channel # from file name
ptr x = val 6.          parameter 6 will hold channel #
to setp by g.           set para. 6 to channel #
val x = w.              put channel # in val x
val w = v.              restore val w
val v = 0.              clear val v
ptr x = w.              restore ptr x
loc ov2.
val i = x - e.
sto a = i.
to 29 if ptr x = 9.
rewind i.
to 98 if flg i ne 0.
loc 29.
get x = v.
to 05 if val x = 0.
to 05 if flg x = 3.
ptr y = x + 0.
read next i.
to 98 if flg i ne 0.
loc 30.
to 05 if val x = 0.
val x = x - 1.
get y = y.
val z = char.
to 30 if val y = z.
write next w.
to 29 if flg w = 0.
stop.
loc 33.
get y = v.
to 23 if flg y = 3.
to 05 if val y = 0.
get x = y.
flg f = 2.              set flag for not adding macro definition
flg b = 2.
get w = f.
ptr z = f + 0.
to 58 by b.
flg f = 0.              reset flag
flg b = 0.
flg w = y.
ptr w = u + 0.
ptr z = y + 0.
```

```
ptr v = v + 7.
get y = v.
to 23 if flg y = 3.
ptr x = y + 0.
flg z = 1.
val z = y + 0.
to 35 if flg w ne 1.
sto w = z.
to 05 if val y = 0.
loc 34.
get x = x.
ptr w = z + 0.
get z = w.
val z = x + 0.
sto w = z.
val y = y - 1.
to 35 if ptr z = 0.
to 34 if val y ne 0.
to 05.
loc 35.
ptr z = 8 + 0.
sto w = z.
ptr 8 = 8 + 7.
to 97 if ptr 8 ge 9.
ptr w = z + 0.
get z = x.
ptr x = z + 0.
val y = y - 1.
to 35 if val y ne 1.
sto w = 0.
to 05.
loc 36.
get i = k.
ptr k = k + 7.
get y = v.
ptr v = v + 7.
get z = v.
to 23 if flg y = 3.
to 23 if flg z = 3.
ptr v = v + 7.
to 41 if val y ne z.
to 38 if val y = 0.
ptr x = z + 0.
loc 37.
get x = x.
get y = y.
to 41 if val x ne y.
val z = z - 1.
```

```
to 37 if val z ne 0.
loc 38.
to 05 if val i ne e.
to 42.
loc 39.
get i = k.
ptr k = k + 7.
get y = v.
to 23 if flg y = 3.
to 74 by p.
ptr i = n + 0.
ptr v = j + 4.
get y = v.
to 23 if flg y = 3.
to 74 by p.
ptr v = j + 4.
ptr v = v + 7.
ptr n = n - i.
to 38 if ptr n = 0.
to 40 if ptr n ge 0.
to 05 if val i = o.
to 41.
loc 40.
to 05 if val i = n.
loc 41.
to 05 if val i = e.
loc 42.
get y = v.
to 23 if flg y = 3.
to 05 if val y = 0.
to 74 by p.
ptr m = n + 0.
to 05.
loc 43.
ptr y = c - 9.
ptr y = y / 7.
val y = ptr y.
to 07 if val y = 0.
ptr y = c + 0.
to 74 by p.
flg y = 1.
val y = 0 + 0.
ptr y = n + 1.
sto c = r.
ptr z = r + 0.
ptr r = c + 0.
ptr c = c - 4.
sto c = k.
```

```
loc 44.
ptr c = r + 0.
ptr r = z + 0.
ptr y = y - 1.
to 05 if ptr 0 ge y.
ptr r = c + 0.
ptr c = c - 7.
sto c = y.
ptr c = c - 7.
get k = c.
ptr c = c - 7.
to 05.
loc 45.
sto 9 = 1.
ptr w = c - 9.
ptr w = w / 7.
flg w = 0.
val w = ptr w.
ptr w = c + 0.
ptr 9 = 9 - 7.
flg b = 2.
ptr b = 0 + 0.
flg u = 0.
val u = r + 0.
ptr u = 7 + 0.
flg z = 1.
val z = 0 + 0.
ptr z = 0 + 0.
ptr x = 9 - 7.
loc 46.
val z = z + 1.
sto 9 = z.
ptr 9 = 9 - 7.
sto 9 = u.
ptr 9 = 9 - 7.
sto 9 = b.
ptr 9 = 9 - 7.
ptr k = k + 7.
get i = k.
ptr i = x - 9.
sto 9 = i.
ptr x = 9 + 0.
ptr 9 = 9 - 7.
to 97 if ptr 8 ge 9.
to 46 if flg i ne 1.
sto 9 = b.
flg b = 0.
ptr z = 9 + 0.
```

```
ptr 9 = 9 - 7.
val u = m + 0.
sto 9 = u.
ptr 9 = 9 - 7.
sto 9 = r.
ptr 9 = 9 - 7.
sto 9 = c.
ptr 9 = 9 - 7.
sto 9 = v.
ptr 9 = 9 - 7.
sto 9 = y.
ptr r = 9 - 7.
sto r = z.
ptr 9 = r - 7.
sto 9 = w.
ptr 9 = 9 - 4.
sto 9 = k.
ptr z = z - 7.
to 48.
loc 47.
to 05 if ptr r = 0.
get z = r.
loc 48.
ptr u = r - 7.
get y = u.
to 44 if flg y = 1.
to 49 if val y = 0.
sto u = 0.
ptr u = u - 4.
get k = u.
ptr v = u + 0.
ptr 9 = u - 7.
ptr c = 9 + 0.
get x = y.
to 99 by b.
ptr y = r + 4.
get w = y.
ptr y = r - 4.
to 97 if ptr 8 ge y.
get y = y.
sto w = y.
to 05.
loc 99.
to 60 if val z ne 1.
flg x = 0.
val x = y - 1.
val y = 1 + 0.
ptr u = u + 7.
```

```
sto u = y.
ptr u = u + 7.
sto u = x.
return by b.
loc 49.
to 44 if flg y = 1.
ptr r = r + 7.
get y = r.
ptr r = r + 7.
get w = r.
sto w = y.
ptr r = r + 7.
get c = r.
ptr r = r + 7.
get r = r.
to 05.
loc 50.
flg y = 1.
val y = 1 + 0.
ptr 8 = 8 - 7.
to 54.
loc 51.
val i = char.
sto 8 = i.
to 52 if val i = c.
to 52 if val i = d.
val i = i - e.
flg z = 3.
val z = char.
val z = z - e.
ptr z = val i.
sto 8 = z.
to 52 if ptr 0 ge z.
to 52 if ptr z ge 5.
flg z = 2.
ptr z = z * 7.
sto 8 = z.
loc 52.
ptr 8 = 8 + 7.
to 97 if ptr 8 ge 9.
val i = char.
sto 8 = i.
to 51 if val i = d.
to 53 if val i = 1.
to 52 if val i ne c.
loc 53.
ptr y = 8 + 0.
sto u = y.
```

```
ptr u = 8 + 0.
loc 54.
get i = a.
read next i.
to 98 if flg i ne 0.
val i = char.
ptr i = 0 + 0.
ptr 8 = 8 + 7.
sto 8 = i.
to 51 if val i = d.
to 52 if val i ne c.
ptr y = 8 + 0.
sto u = y.
sto 8 = 1.
ptr 8 = 8 + 7.
to 97 if ptr 8 ge 9.
val i = char.
to 55 if val i ne c.
flg b = 0.
to addef if flg e = 2.  test add flag
loc 55.
return by d.
loc addef.              return from adding a definition
flg e = 0.              reset add flag
get i = a.              restore input unit
val i = h + 0.
sto a = i.
ptr d = o.              restore d's previous return address
to 55.
loc 56.
val w = 3 + 0.
ptr x = c + 0.
loc 57.
get x = x.
char = val x.
to 57 if flg x ne 1.
write next w.
to 98 if flg w ne 0.
to 55 if val x = 1.
char = val x.
to 57.
loc 58.
ptr z = w + z.
to 60 if ptr w ne 0.
to 71 if flg b = 2.
loc 59.
to 70 if ptr v ge 9.
get z = v.
```

```
get y = q.
get x = y.
to 63 if flg z = 2.
to 64 if flg z = 3.
ptr v = q + 7.
ptr q = v + 7.
loc 60.
get w = z.
to 69 if flg w = 1.
to 62 if flg w = 2.
to 58 if val y = 0.
to 58 if val x ne w.
to 61 if ptr w = 0.
to 61 if flg x = 3.
to 61 if flg b = 2.
ptr q = v - 7.
ptr v = q - 7.
to 97 if ptr 8 ge v.
sto q = y.
ptr w = w + z.
sto v = w.
loc 61.
val y = y - 1.
ptr y = x + 0.
get x = x.
ptr z = z + 7.
to 60.
loc 62.
to 61 if flg x = 2.
to 58 if flg b = 2.
ptr q = v - 7.
ptr v = q - 7.
to 97 if ptr 8 ge v.
sto q = y.
flg z = 2.
sto v = z.
flg x = 3.
to 58.
loc 63.
flg z = 3.
ptr z = z + 7.
sto v = z.
ptr u = u + 7.
flg w = 0.
val w = 0 + 0.
ptr w = y + 0.
sto u = w.
to 60.
```

```
loc 64.
to 68 if val y = 0.
to 68 if val x = r.
get w = u.
val w = w + 1.
val y = y - 1.
ptr y = x + 0.
to 67 if val x ne m.
val z = 0 + 0.
loc 65.
val z = z + 1.
loc 66.
to 68 if val y = 0.
get x = x.
val y = y - 1.
ptr y = x + 0.
val w = w + 1.
to 65 if val x = m.
to 66 if val x ne r.
val z = z - 1.
to 66 if val z ne 0.
loc 67.
get x = x.
sto q = y.
sto u = w.
to 60.
loc 68.
sto u = 3.
ptr u = u - 7.
ptr v = q + 7.
ptr q = v + 7.
to 59.
loc 69.
to 58 if val y ne 0.
ptr u = z + 7.
get y = u.
to 70 if flg b ne 2.
to 70 if flg f = 2.
ptr g = u + 7.
ptr w = 8 - 7.
flg w = val 4.
sto g = w.
loc 70.
return by b.
loc 71.
ptr w = 8 - z.
sto z = w.
to 73 if val y = 0.
```

```
loc 72.
val y = y - 1.
ptr y = x + 0.
ptr x = 0 + 0.
sto 8 = x.
ptr 8 = 8 + 7.
to 97 if ptr 8 ge 9.
get x = y.
to 72 if val y ne 0.
loc 73.
flg x = 1.
ptr x = 0 + 0.
sto 8 = x.
ptr u = 8 + 7.
flg y = 0.
ptr y = u + 0.
sto u = y.
ptr 8 = u + 7.
to 97 if ptr 8 ge 9.
return by b.
loc 74.
ptr o = 9 + 0.
val s = y + 0.
ptr s = y + 0.
ptr t = 0 + 0.
to 75 if val y ne 0.
ptr n = 0 + 0.
return by p.
loc 75.
val t = m + 0.
loc 76.
to 93 if val s = 0.
get x = s.
ptr y = s + 0.
val y = 0 + 0.
to 77 if val x ne m.
sto 9 = t.
ptr 9 = 9 - 7.
to 97 if ptr 8 ge 9.
val s = s - 1.
ptr s = x + 0.
to 75.
loc 77.
to 78 if val x = n.
to 78 if val x = o.
to 78 if val x = p.
to 78 if val x = q.
to 78 if val x = r.
```

```
val y = y + 1.
get x = x.
to 77 if val s ne y.
val x = r + 0.
val s = s + 1.
loc 78.
val j = x + 0.
ptr n = 0 + 0.
val s = s - y.
val s = s - 1.
ptr s = x + 0.
to 83 if val y = 0.
get x = y.
ptr u = val x.
ptr u = u - e.
to 79 if ptr u ge 5.
to 81 if ptr u ge 0.
loc 79.
ptr v = 9 + 7.
get w = f.
flg y = 0.
ptr z = f + 0.
to 58 by b.
to 83 if flg y ne 1.
to 83 if val y = 0.
get x = y.
flg n = 1.
to 82 if val x = o.
flg n = 0.
ptr x = y + 0.
loc 80.
get x = x.
ptr u = val x.
ptr u = u - e.
to 81 if ptr u = 0.
to 93 if ptr u ge 5.
to 93 if ptr 0 ge u.
loc 81.
ptr n = n * 5.
ptr n = n + u.
loc 82.
val y = y - 1.
to 80 if val y ne 0.
to 83 if flg n = 0.
flg n = 0.
ptr n = 0 - n.
loc 83.
to 92 if val j = r.
```

```
to 90 if val t = m.
to 89 if val j = p.
to 89 if val j = q.
loc 84.
to 87 if val t = q.
to 86 if val t = p.
to 85 if val t = o.
ptr t = t + n.
to 88.
loc 85.
ptr t = t - n.
to 88.
loc 86.
ptr t = t * n.
to 88.
loc 87.
ptr t = t / n.
loc 88.
val t = j + 0.
to 76 if val j ne r.
ptr n = t + 0.
ptr 9 = 9 + 7.
get t = 9.
to 92.
loc 89.
to 86 if val t = p.
to 87 if val t = q.
loc 90.
sto 9 = t.
ptr 9 = 9 - 7.
to 97 if ptr 8 ge 9.
val t = j + 0.
ptr t = n + 0.
to 76.
loc 91.
to 93 if val s ne 0.
return by p.
loc 92.
to 84 if val t ne m.
to 91 if ptr 9 = o.
ptr 9 = 9 + 7.
get t = 9.
to 92 if val s = 0.
get x = s.
val s = s - 1.
ptr s = x + 0.
val j = x + 0.
to 92 if val j = r.
```

```
to 83 if val j = n.
to 83 if val j = o.
to 83 if val j = p.
to 83 if val j = q.
loc 93.
message expr to 4.
ptr n = 0 + 0.
ptr 9 = o + 0.
to 94 by b.
return by p.
loc 94.
ptr x = c + 0.
ptr y = j + 0.
to 96 if ptr 9 ge c.
sto 9 = 1.
loc 95.
get x = x.
char = val x.
to 95 if flg x = 0.
write next 4.
to 98 if flg 4 ne 0.
to 96 if val x = 1.
char = val x.
to 95.
loc 96.
to 70 if flg p = 1.    if tracing only trace back one call
to 70 if ptr y = 0.
ptr y = y - h.
ptr x = y - 7.
get y = y.
get x = x.
to 95.
loc 97.
message full to 4.
to 94 by b.
stop.
loc 98.
message ioch to 4.
to 94 by b.
stop.
loc tr.                tracing routine
message trace to 4.    print tracing message on terminal
to 94 by b.            call error traceback
return by g.
end program.
```

APPENDIX 7.3

FLUB TO C MACROS FOR ASP

Following are the macros used to translate the FLUB version of ASP to the C programming language. Either STAGE2 or ASP can use the macros to do the translation.

```
 .'$'0 (+-*/)
' ' = '.
,get letter 1 of '10 in 11$
11'86$
,on '10 = val set hv to 57 else 51$
,if '38 gt hv skip 2$
        j'81'20 = '30;'f1$
'f9$
        j'81'20 = j'81'30;'f1$
$
' ' = ' + '.
,get letter 1 of '10 in 11$
11'86$
,on '10 = val set hv to 57 else 51$
,if '38 gt hv skip 4$
,if '48 gt hv skip 6$
'30+'40'96$
        j'81'20 = '94;'f1$
'f9$
,if '48 neq 48 skip 5$
        j'81'20 = j'81'30;'f1$
'f9$
,if '38 neq 48 skip 2$
        j'81'20 = j'81'40;'f1$
'f9$
        j'81'20 = j'81'30 + j'81'40;'f1$
$
' ' = ' - '.
,get letter 1 of '10 in 11$
11'86$
,on '10 = val set hv to 57 else 51$
,if '38 gt hv skip 4$
,if '48 gt hv skip 6$
'30-'40'96$
```

```
        j'81'20 = '94;'f1$
'f9$
,if '48 neq 48 skip 2$
        j'81'20 = j'81'30;'f1$
'f9$
        j'81'20 = j'81'30 - j'81'40;'f1$
$
' ' = ' ' '.
,get letter 1 of '10 in l1$
l1'86$
,on '10 = val set hv to 57 else 51$
,if '38 gt hv skip 4$
,if '58 gt hv skip 3$
'30'40'50'96$
        j'81'20 = '94;'f1$
'f9$
        j'81'20 = j'81'30 '40 j'81'50;'f1$
$
' ' = ' '.
,get letter 1 of '10 in l1$
l1'86$
,on '30 = val set hv to 57 else 51$  implementation dependent
,if '48 gt hv skip 2$
        j'81'20 = '40;'f1$
'f9$
,get letter 1 of '30 in l2$
l2'76$
        j'81'20 = j'71'40;'f1$
$
get ' = '.
        jf'10 = (l[jp'20+1] >> 24) & ~(~0 << 8);'f1$
        jv'10 = l[jp'20+1] & 077777777;'f1$
        if (((jv'10 >> 23) & ~(~0 << 1)) == 1) jv'10 = -1;'f1$
        jp'10 = l[jp'20];'f1$
$
sto ' = '.
        l[jp'10+1] = jf'20 << 24;'f1$
        l[jp'10+1] |= (jv'20 & 077777777);'f1$
        l[jp'10] = jp'20;'f1$
$
to ' if ' ' = '.
,if '30 ne '40 skip 2$
        goto k'10;'f1$
'f9$
,get letter 1 of '20 in l1$
l1'86$
,on '20 = val set hv to 57 else 51$
,if '38 gt hv skip 2$
```

```
        if ('30 == j'81'40) goto k'10;'f1$
'f9$
,if '48 gt hv skip 2$
        if (j'81'30 == '40) goto k'10;'f1$
'f9$
        if (j'81'30 == j'81'40) goto k'10;'f1$
$
to ' if ' ' ne '.
,if '30 = '40 skip 10$
,get letter 1 of '20 in l1$
l1'86$
,on '20 = val set hv to 57 else 51$
,if '38 gt hv skip 2$
        if ('30 != j'81'40) goto k'10;'f1$
'f9$
,if '48 gt hv skip 2$
        if (j'81'30 != '40) goto k'10;'f1$
'f9$
        if (j'81'30 != j'81'40) goto k'10;'f1$
$
to ' if ptr ' ge '.
,if '20 ne '30 skip 2$
        goto k'10;'f1$
'f9$
,if '28 gt 51 skip 2$
        if ('20 >= jp'30) goto k'10;'f1$
'f9$
,if '38 gt 51 skip 2$
        if (jp'20 >= '30) goto k'10;'f1$
'f9$
        if (jp'20 >= jp'30)  goto k'10;'f1$
$
to ' by '.
        jp'20 = '00;'f1$
        goto k'10;'f1$
99+'00'96$
 k'94:'f1$
$
return by '.
        switch   (jp'10) { 'f1$
        case 1:  goto k100;'f1$
        case 2:  goto k101;'f1$
        case 3:  goto k102;'f1$
        case 4:  goto k103;'f1$
        case 5:  goto k104;'f1$
        case 6:  goto k105;'f1$
        case 7:  goto k106;'f1$
        case 8:  goto k107;'f1$
```

```
        case 9:  goto k108;'f1$
        case 10: goto k109;'f1$
        case 11: goto k110;'f1$
        case 12: goto k111;'f1$
        case 13: goto k112;'f1$
        case 14: goto k113;'f1$
        case 15: goto k114;'f1$
        case 16: goto k115;'f1$
        case 17: goto k116;'f1$
        case 18: goto k117;'f1$
        case 19: goto k118;'f1$
        case 20: goto k119;'f1$
        case 21: goto k120;'f1$
        case 22: goto k121;'f1$
        case 23: goto k122;'f1$
        case 24: goto k123;'f1$
        case 25: goto k124;'f1$
        case 26: goto k125;'f1$
        case 27: goto k126;'f1$
        case 28: goto k127;'f1$
        case 29: goto k128;'f1$
        case 30: goto k129;'f1$
                        }  'f1$
$
to '.
        goto k'10;'f1$
$
stop.
        goto k992;'f1$
$
val ' = char.
        jv'10 = lb[lbr];'f1$
        lbr = lbr+1;'f1$
$
char = val '.
        iwrch (jv'10,lb,&lbw,&jf'10,&lbl);'f1$
$
read next '.
        jf'10 = ioop (-1,jv'10,lb,1,&lbl);'f1$
        lb[lbl] = -1;'f1$
        lbr = 1;'f1$
$
write next '.
        jf'10 = ioop (1,jv'10,lb,1,&lbl);'f1$
        lbw = 1;'f1$
$
rewind '.
        jf'10 = ioop (0,jv'10,lb,1,&one);'f1$
```

```
        jf'10 = 0;'f1$
$
loc '.
 k'10:'f1$
$
message full to '.
        mb[11] = 102;'f1$
        mb[12] = 117;'f1$
        mb[13] = 108;'f1$
        mb[14] = 108;'f1$
        jf'10 = ioop (1,jv'10,mb,1,&twenty1);'f1$
$
message ioch to '.
        mb[11] = 105;'f1$
        mb[12] = 111;'f1$
        mb[13] = 99;'f1$
        mb[14] = 104;'f1$
        jf'10 = ioop (1,jv'10,mb,1,&twenty1);'f1$
$
message conv to '.
        mb[11] = 99;'f1$
        mb[12] = 111;'f1$
        mb[13] = 110;'f1$
        mb[14] = 118;'f1$
        jf'10 = ioop (1,jv'10,mb,1,&twenty1);'f1$
$
message expr to '.
        mb[11] = 101;'f1$
        mb[12] = 120;'f1$
        mb[13] = 112;'f1$
        mb[14] = 114;'f1$
        jf'10 = ioop (1,jv'10,mb,1,&twenty1);'f1$
$
message trace to '.
        printf("*** trace *** ");'f1$
        fflush(stdout);'f1$
$
end program.
 k992:  ; }'f1$
'f0$
$
message * error.
        for (j = 1; j <= 9; j++)'f1$
                    mb[j] = 42;'f1$
        mb[10] = 32;'f1$
        mb[15] = 32;'f1$
        mb[16] = 101;'f1$
        mb[17] = 114;'f1$
```

```
        mb[18] = 114;'f1$
        mb[19] = 111;'f1$
        mb[20] = 114;'f1$
$
call clil.  subroutine to execute one operating system command
        os(lb,1,lbl);'f1$
$
call cli.
        printf("Escaping to the shell.\n");'f1$
        printf("To re-enter ASP hit control-d.\n");'f1$
        system("sh");'f1$
        printf("Goodbye to the shell.  Re-entering ASP.\n");'f1$
$
call barreled.
        printf("call barreled not implemented\n");'f1$
$
getch ' in w.  returns channel # for file name
        jf'10 = ioop(2,0,lb,jv'10,&jvw);'f1$
$
stochanp6.
        jpz = memla;'f1$
        l[memla+1] = jvw;'f1$
        jvz = 1;'f1$
$
stochanp8.
        jpz = memla+2;'f1$
        l[memla+3] = jvw;'f1$
        jvz = 1;'f1$
$
close next '.
        jf'10 = ioop(3,jv'10,lb,1,&lbl);'f1$
$
gigi.
        jfw = gigi(jvw,lb,1,lbl);'f1$
        lbw = 1;'f1$
$
.       null macro to allow comments in flub program
$
.       the remaining macro definitions are "system macros"
$       in that they just provide convenient access to stage2
.       functions i.e. these are not flub operations.
$
text '.           switch the input channel
'10'26 '16$
'20'f2$
$
,sto '='.         store a value into memory
'f3$
```

```
$
,set ' = '.          store an integer into memory
'24'26 'f3$
$
,skip '.             skip lines unconditionally
'f4$
$
,if ' = ' skip '.
'f50$
$
,ifc ' = ' skip '.
'11'16 'f50$
$
,if ' ne ' skip '.
'f51$
$
,if ' lt ' skip '.
'f6-$
$
,if ' eq ' skip '.
'f60$
$
,if ' gt ' skip '.
'f6+$
$
,if ' neq ' skip '.
'f61$
$
,if ' le ' skip '.
,if '10 eq '20 skip '34+1$
'f6-$
$
,if ' ge ' skip '.
,if '10 eq '20 skip '34+1$
'f6+$
$
.
$   get 1st letter of arg. 1 and store in arg. 2
,get letter 1 of ' in '.
'10'17$
,sto '20='10$
,skip 1$
'f8$
$
,on ' = ' set ' to ' else '.
,if '10 = '20 skip 2$
,sto '30='50$
'f9$
```

```
,sto '30='40$
$
. init is necessary so that the initialization stuff won't
$  match other macros
init'.
'10'f1$
$$
. #include isn't part of an init macro since it contains
. source-eol symbol (".")
#include <stdio.h>
init#define memla 39996
init                /* memla points to 4 spaces added to the end
init                of array 1 where we put the channel numbers
init                gotten from the file names when using *
init                with certain stage2 functions;  the first 2
init                spaces are for parameter 6 and the last 2
init                spaces are for parameter 8 */
init
init    int a,b,c,e;
init
init    main(argc,argv)
init     int argc;
init     char *argv[];
init  {
init
init    long j,lb[82],lbl,lbr,lbw,mb[21],one,twenty1;
init    long jf0,jf1,jf2,jf3,jf4,jf5,jf6,jf7,jf8,jf9,jfa,jfb,jfc,jfd;
init    long jfe,jff,jfg,jfh,jfi,jfj,jfk,jfl,jfm,jfn,jfo,jfp,jfq,jfr;
init    long jfs,jft,jfu,jfv,jfw,jfx,jfy,jfz;
init    long jv0,jv1,jv2,jv3,jv4,jv5,jv6,jv7,jv8,jv9,jva,jvb,jvc,jvd;
init    long jve,jvf,jvg,jvh,jvi,jvj,jvk,jvl,jvm,jvn,jvo,jvp,jvq,jvr;
init    long jvs,jvt,jvu,jvv,jvw,jvx,jvy,jvz;
init    long jp0,jp1,jp2,jp3,jp4,jp5,jp6,jp7,jp8,jp9,jpa,jpb,jpc,jpd;
init    long jpe,jpf,jpg,jph,jpi,jpj,jpk,jpl,jpm,jpn,jpo,jpp,jpq,jpr;
init    long jps,jpt,jpu,jpv,jpw,jpx,jpy,jpz;
init    long l[40000];
init
init    if  (argc != 5) { printf("Wrong number of arguments: adios\n");
init                      exit(1); }
init    a = open(argv[1],0);
init    b = creat(argv[2],0755);
init    close(b);  /* saves file descriptors */
init    b = open(argv[2],2);
init    c = creat(argv[3],0755);
init    close(c);  /* saves file descriptors */
init    c = open(argv[3],1);
init    e = open(argv[4],0);
init    one = 1;
```

```
init     twenty1 = 21;
init      jp9 = 39995;
init      jf0 = 0;
init      jf1 = 1;
init      jf2 = 2;
init      jf3 = 3;
init      jv0 = 0;
init      jv1 = 1;
init      jv2 = 2;
init      jv3 = 3;
init      jv4 = 4;
init      jv5 = 5;
init      jv6 = 6;
init      jv7 = 7;
init      jv8 = 8;
init      jv9 = 9;
init      jp0 = 0;
init      jp1 = 1;
init      jp2 = 2;
init      jp3 = 3;
init      jp5 = 10;
init      jp7 = 2;
init      jp8 = 1;
init      lbl = 1;
init      lbr = 1;
init      lbw = 1;
init      jp9 = jp8 + (jp9 / jp7 - 1) * jp7;
message * error.
text 5.
```

APPENDIX 7.4

C SUPPORT ROUTINES FOR ASP

Following are listings of the four hand coded support routines used in the C implementation of the ASP processor. These routines implement the machine dependent portions of the ASP processor. The routines are:

gigi.c – provides the interface to the GIGI/Regis graphics terminal

ioop.c – provides the I/O operations

iwrch.c – puts a character in the line buffer

os.c – provides an interface to the operating systems command language interpreter

```
#include <stdio.h>

/*************************************************************/
/* GIGI                                                      */
/* Subroutine to interface with the Gigi/Regis graphics      */
/* terminal.  Four operations can be performed depending     */
/* on the value of ifunc.  Regis commands are passed         */
/* through the ilist parameter.                              */
/*************************************************************/

   gigi(ifunc, ilist, jp1, jp2)
          long ifunc;
          long ilist[];
          long jp1,jp2;
 {
   int i, j;
   char cbuff[80];

   switch (ifunc)
    {
       case 0:  /* open gigi terminal for graphics */
                printf("%cPp\n",'\033');
                break;

       case 1:  /* send graphics command to gigi terminal */
                for (i = jp1, j = 0; i < jp2; i++, j++)
```

```
                cbuff[j] = ilist[i];
            cbuff[--i] = ' ';
            printf("%s\n",cbuff);
            break;

      case 2:  /* close gigi terminal for graphics */
            printf("%c%c\n",'\033','\');
            break;

      case 3:  /* reset graphic attributes */
            /* set terminal to ANSI mode */
            printf("\033PrTM1\033");
            /* clear all graphics attributes */
            /* (only works in ANSI mode) */
            printf("\033c");
            fflush(stdout);
            /* return to VT52 mode */
            system("asp.TM0");
            break;

      default: return(1);  /* bad gigi function */
    }

   return(0);  /* good return */

 }

#include <stdio.h>

/*************************************************************/
/* IOOP                                                      */
/* Subroutine to provide I/O operations for the ASP          */
/* processor.  Five operations can be performed depending    */
/* on the value of ifunc (read, write, rewind, close, and    */
/* associate a file name with a channel number).  The        */
/* channel number is passed through the ifile parameter.     */
/* ilist is the I/O buffer.  Return values are:              */
/* 0 - successful, 1 - end of file, 2 - error                */
/*************************************************************/

   ioop(ifunc, ifile, ilist, jp1, jp2)
         long ifunc, ifile;
         long ilist[];
         long jp1,*jp2;
 {
    char wbuf[81];
    short i,j,index;
    static char rbuf1[256], rbuf2[256], rbuf5[256];
    static short bufin1 = 0, bufin2 = 0, bufin5 = 0;
    static short bufl1 = 0, bufl2 = 0, bufl5 = 0;
```

```
/* buffer index and buffer length for channels 6 - 35 */
static short bufinex[30], buflex[30];
/* extrachannelbufferpointer points to buffers for */
/* channels 6 - 35 */
static char *exchbufp[30];
static int fd[30];  /* file descriptors for channels 6 - 35 */
static short numfop = 0; /* number of files open */
/* extrachannelfilenamepointer points to file name */
/* for channels 6 - 35 */
static char *exchfnp[30];
char fname[80];
short tfile;
char *bufpt, *malloc();
char rbuf4;
short bufind, buflen;
int fdnum;
extern  a,b,c,e;
switch (ifile)
 {
   case 0:
            if (ifunc == -1) return(1);
            if (ifunc == 2) break;
            else return(0);
   case 1:
            fdnum = a;
            bufpt = rbuf1;
            bufind = bufin1;
            buflen = bufl1;
            break;
   case 2:
            fdnum = b;
            bufpt = rbuf2;
            bufind = bufin2;
            buflen = bufl2;
            break;
   case 3:
            fdnum = c;
            break;
   case 4:
            fdnum = 1;   /* terminal output
                            (terminal input handled specially) */
            break;
   case 5:
            fdnum = e;
            bufpt = rbuf5;
            bufind = bufin5;
            buflen = bufl5;
            break;
```

```
    default:
            /* channels a-z are passed as */
            /* 49-74, we make them 10-35 */
            if (ifile >= 49 && ifile <= 74)
               ifile -= 39;
            else if (ifile < 6 || ifile > 9)
               {
printf("error -- unexpected call on unknown file\n");
                  return(2);
               }
            tfile = ifile - 6;
            if (exchbufp[tfile] == NULL)
               {  if (++numfop >= 14)
                     {
printf ("Only 13 extra files can be open at one\n");
printf ("time under the UNIX operating system.\n");
printf ("Request ignored.\n");
                        return(0);
                     }
                  exchbufp[tfile] = malloc(256);
                  bufinex[tfile] = 0;
                  buflex[tfile] = 0;
                  /* convert to a-z or 6-9 */
                  if (ifile > 9) wbuf[0] = ifile + 87;
                  else wbuf[0] = ifile + 48;
                  printf("You have asked to use a new file");
                  printf(" (file number %c).\n",wbuf[0]);
                  printf("Type in its name please.\n");
                  for (i=0; i < 80; i++)
                      {  j = read(0,&fname[i],1);
                         if (fname[i] == '0') break;
                      }
                  fname[i] = ' ';
                  if ((fdnum = open(fname,2)) == -1)
                     if ((fdnum = creat(fname,0755)) == -1)
                        {
printf("error opening file %s\n",fname);
                           exit(1);
                        }
                           /* saves file descriptors */
                     else { close(fdnum);
                            fdnum = open(fname,2);
                          }
                  exchfnp[tfile] = malloc(i+1);
                  strcpy(exchfnp[tfile],fname);
                  fd[tfile] = fdnum;
               }
            else fdnum = fd[tfile];
```

```
            bufpt = exchbufp[tfile];
            bufind = bufinex[tfile];
            buflen = buflex[tfile];
            break;
 }
switch (ifunc)
 {
   case -1:                  /* read operation */
     /* read invalid on channel 3 */
     if (ifile == 3) return(2);
     index = jp1;
     if (ifile == 4)   /* read from the terminal */
        { for (i=0; i<80; i++, index++)
              { if ((j = read(0,&rbuf4,1)) == -1)
                    return(2); /* bad read */
                if (j == 0) return(1);   /* eof return */
                if (rbuf4 == '0') break;   /* end of line */
                ilist[index] = rbuf4;
              }
        }
     else
        { while (index-jp1 < 80)
              { if (bufind >= buflen)
                    { buflen = read(fdnum,bufpt,256);
                      bufind = 0;
                      if (buflen == 0)
                                              /* eof */
                         if (index == jp1) return(1);
                         else break;  /* next read is eof */
                    }
                /* end of line */
                if (*(bufpt + bufind) == '0') break;
                ilist[index++] = *(bufpt + bufind++);
              }
          switch (ifile)
           {
             case 1:
                      bufin1 = bufind + 1;
                      bufl1 = buflen;
                      break;
             case 2:
                      bufin2 = bufind + 1;
                      bufl2 = buflen;
                      break;
             case 5:
                      bufin5 = bufind + 1;
                      bufl5 = buflen;
```

```
                    break;
            default:
                    bufinex[tfile] = bufind + 1;
                    buflex[tfile] = buflen;
                    break;
          }
      }
    *jp2 = index;  /* point to next free space */
    return(0);              /* "good return" */

  case 0:                   /* rewind operation */

    lseek (fdnum, (long) 0, 0);
    switch (ifile)
     {
       case 1:
                bufin1 = 0;
                buf11 = 0;
                break;
       case 2:
                bufin2 = 0;
                buf12 = 0;
                break;
       case 3:
                /* rewind invalid on channel 3 */
                return(2);
       case 4:
                break;
       case 5:
                bufin5 = 0;
                buf15 = 0;
                break;
       default:
                bufinex[tfile] = 0;
                buflex[tfile] = 0;
                break;
     }
    return(0);              /* "good return" */

  case 1:                   /* write operation */

    for (index = jp1, i = 0; index < *jp2 && i < 80;
         index++, i++)
        wbuf[i] = ilist[index];
    if (i == 80) --i;
       else wbuf[i] = '0';
    /* because of buffering file pointer */
    if (ifile != 4 && ifile != 3)
         /* is not in proper position for write */
```

```
    { if (buflen != bufind)
        { lseek (fdnum, (long) bufind-buflen, 1);
          /* move buffer index past what was */
          /* written over */
          switch (ifile)
           { case 1:  bufin1 = bufind+i+1;
                      break;
             case 2:  bufin2 = bufind+i+1;
                      break;
             case 5:  bufin5 = bufind+i+1;
                      break;
             default: bufinex[tfile] = bufind+i+1;
                      break;
           }

        }
    }
  j = write(fdnum,wbuf,i+1);
  if (j <= 0) return(2);   /* bad write */
  /* move file pointer back past buffer */
  if (ifile != 4 && ifile != 3)
     if (buflen != bufind)
        lseek (fdnum, (long) buflen-bufind-i-1, 1);
  return(0);              /* "good return" */

case 2:     /* return a channel for a file name */

  if (ifile != 0) return(2);
  for (i=1; i<=jp1; i++)
      fname[i-1] = ilist[i];
  fname[i-1] = ' ';
  for (i=0; i<30; i++)
      { if (strcmp(fname,exchfnp[i]) == 0)
          if (exchbufp[i] != NULL)
              /* convert from 0-29 to 6-35 */
            { *jp2 = i+6;
              /* convert to a-z or 6-9 */
              if (*jp2 > 9) *jp2 += 87;
              else *jp2 += 48;
              return(0);

            }
      }
  for (i=0; i<30; i++)
      { if (exchbufp[i] == NULL)
          { exchbufp[i] = malloc(256);
            bufinex[i] = 0;
            buflex[i] = 0;
            if ((fdnum = open(fname,2)) == -1)
```

```
                    if ((fdnum = creat(fname,0755)) == -1)
                       {
    printf("error opening file %s\n",fname);
                         exit(1);
                       }
                    else { close(fdnum);
                           fdnum = open(fname,2);
                         }
                  exchfnp[i] = malloc(strlen(fname)+1);
                  strcpy(exchfnp[i],fname);
                  fd[i] = fdnum;
                  *jp2 = i+6;  /* convert from 0-29 to 6-35 */
                  /* convert to a-z or 6-9 */
                  if (*jp2 > 9) *jp2 += 87;
                  else *jp2 += 48;
                  return(0);
                }
          }
        printf("all available channels in use; request ignored\n");
        return(0);

      case 3:     /* close operation */

        if (ifile < 6 || ifile > 35) return(2);
        if (exchbufp[tfile] == NULL)
           { printf("channel %d not open; close request ignored\n",
                  ifile);
             return(0);
           }
        free(exchbufp[tfile]);
        free(exchfnp[tfile]);
        exchbufp[tfile] = NULL;
        close(fdnum);
        return(0);

      default:

        printf("error --> invalid ioop function requested");
        exit(1);

    }       /* end of switch on ifunc */
 }   /* end of ioop */

/*************************************************************/
/* IWRCH                                                     */
/* Subroutine to insert a character (jchar) into the ASP     */
/* line buffer preparing for output.   If the character      */
/* is less than zero or the buffer is full (max 80 chars)    */
/* then the buffer is closed with a -1.  jbuff is the        */
/* buffer.  jindx is the position to insert the character.   */
```

```
/* jflag = 0 means the character was inserted, jflag = 1  */
/* means the buffer was closed.  jleng is the number of   */
/* characters in the buffer when it is closed.            */
/*********************************************************/

 iwrch(jchar,jbuff,jindx,jflag,jleng)
 long jchar,jbuff[],*jindx,*jflag,*jleng;
 {
    if (jchar < 0) goto l1;
    if (80 < *jindx) goto l1;
    jbuff[*jindx] = jchar;
    *jindx = *jindx + 1;
    *jflag = 0;
    goto ldone;

l1: jbuff[*jindx] = -1;
     *jleng = *jindx;
     *jindx = 1;
     *jflag = 1;

ldone:
                        ;}

#include <stdio.h>

/*********************************************************/
/* OS                                                    */
/* Subroutine to interface with the operating system.    */
/* One operating system command (passed through the ilist */
/* parameter) is executed.                               */
/*********************************************************/

   os(ilist, jp1, jp2)
          long ilist[];
          long jp1,jp2;
 {
   int i,j;
   char cbuff[80];

   for (i = jp1, j = 0; i < jp2; i++, j++)
       cbuff[j] = ilist[i];
   cbuff[--i] = ' ';
   system(cbuff);
   return;
 }
```

APPENDIX 9.1

BARREL/ASP BSYS KIT

Informal description of the general purpose system commands available in the bsys kit. Keywords are lower case; non–terminals are upper case. A "general case" example is followed by a specific example.

1. lorv(#) – determine if argument is a literal or variable; return literal or value of variable as value of variable %da
   example: lorv(VAL) or lorv(ARRAY)
   lorv('some literal text)
   lorv(array!'3')
   lorv(array!index)
   lorv(avar)

2. if # skip # – if the boolean expression is true the skip the next n lines; equal and not equal only are supported for alphanumeric comparisons
   example: if VAL BOOLOP VAL skip EXP
   if EXP BOOLOP EXP skip EXP
   if var eq 'text' skip 3
   if var ne 'text' skip 3
   if a < > 2 skip 10
   if a = 2 skip 10
   if a < 2 skip 10
   if a > 2 skip 10
   if a < = 2 skip 10
   if a = > 2 skip 10

3. skip # – skip lines unconditionally
   example: skip EXP
   skip 5

4. set # to # – set the value of a variable to a constant
   example: set VARIABLE to LIT
   set var to some text

5. setv # to # – set the value of a variable to the value of a variable
   example: setv VARIABLE to VARIABLE
   setv var1 to var2

6. setx # to # – set the value of a variable to the value of an arithmetic expression
   example: setx VARIABLE to EXP
   setx var to 4*a+3/b-(pi/180)

7. rem# – allow for comments in programs
   example: remLIT
   rem this is a comment

8. pop # – pop from the system stack onto a list of variables
   example: pop VARIABLE [VARIABLE ...]
   pop var1 var2 var3

9. push # – push a list of variables or literals onto the system stack
   example: push VAL [VAL ...]
   push var1 'literalvalue' var2

10. repos # – reposition file pointer past a line beginning with the argument
    example: repos LIT
    repos label1:

11. ty # – output some text to the terminal
    example: ty LIT
    ty hello world

12. text# – start taking input from the specified channel
    example: textCHANNEL
    text5

Definition of non-keywords used in examples above:

CHANNEL	a variable or quoted literal value which evaluates to a channel number (0-9 or a-z) NOTE: channels 0-5 are reserved by the system for specific purposes and may not work with some commands
VAL	either a variable or a quoted literal value
VARIABLE	a variable name which can consist of any sequence of characters which are balanced with respect to parentheses

BOOLOP	a boolean operator; can be any of: eq (string equality) ne (string inequality) = (equal) < > (not equal) < (less than) > (greater than) < = (less than or equal) = < (less than or equal) > = (greater than or equal) = > (greater than or equal) NOTE: eq and ne assume their arguments are strings and the other relational operators assume their arguments are integers
EXP	an arithmetic expression which evaluates to an integer NOTE: an arithmetic expression can involve the four arithmetic operations +, –, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)
ARRAY	an array reference which consists of a variable name followed by the "plink" operator (!) followed by a value (i.e. VARIABLE!VAL)
LIT	a literal constant value

APPENDIX 9.2

BARREL/ASP BBAS KIT

Informal description of the commands available in the bbas kit. Keywords are lower case; non–terminals are upper case. A "general case" example is followed by a specific example.

1. fty – output to the terminal the value of the variable or the quoted value; if there is more than one argument (separated by commas) concatenate them before output
   example: (ty VAL[,VAL...])
   (fty 'the value of var = ,var)

2. readch – get an input value from a channel and make it the new value of the variable
   example: (readch CHANNEL VARIABLE)
   (readch 'a' var)

3. read – get an input value from a file and make it the new value of the variable
   example: (read FILE VARIABLE)
   (read 'infile' var)

4. writech – write a value onto a channel
   example: (writech CHANNEL VAL)
   (writech '7' 'a string)

5. write – write a value onto a file
   example: (write FILE VAL)
   (write 'outfile' 'another string)

6. rewindch – rewind a channel
   example: (rewindch CHANNEL)
   (rewindch 'z)

7. rewind – rewind a file
   example: (rewind FILE)
   (rewind 'afile)

8. closech – close a channel (disassociate the file with the channel)
   example: (closech CHANNEL)
   (closech '6)

9. close – close a file
   example: (close FILE)
   (close 'file1)

10. exefilech – transfer execution to a point within a channel (start executing commands from that channel) depending on EXOP
    example: (exefilech CHANNEL EXOP)
    (exefilech 'a' '+)

11. exefile – transfer execution to a point within a file (start executing commands from that file) depending on EXOP
    example: (exefile FILE EXOP)
    (exefile 'cmdfile' '+)

12. return – return execution to the channel which last issued an exefile command
    example: (return)

13. for/endfor – looping control structure; loop zero or more times depending on incrementing a variable and a terminal value
    example: (for VARIABLE := EXP to EXP begin)
    STATEMENTS
    (end for)
    example: (for VARIABLE := EXP downto EXP begin)
    STATEMENTS
    (end for)
    example: (for i := 1 to 100 begin)
    (ty array!i)
    (end for)

14. stop – stop execution of the program
    example: stop

15. xarray – execute the elements of an array as if they were commands; the zero element must contain the number of elements to execute
    example: (xarray ARRAY)
    (xarray arr)

16. tya – output the contents of an array on the users terminal; the zero element must contain the number of elements to output

example: (tya ARRAY)
(tya arr)

17. wrach – write the contents of an array on a channel; the zero element must contain the number of elements to output
example: (wrach CHANNEL ARRAY)
(wrach 'j' arr)

18. wra – write the contents of an array to a file; the zero element must contain the number of elements to output
example: (wra FILE ARRAY)
(wra 'outfile' arr)

19. getach – read values from a channel and put them in an array; the first line read must be the number of values to read
example: (getach CHANNEL ARRAY)
(getach 'j' arr)

20. geta – read values from a file and put them in an array; the first line read must be the number of elements to read
example: (geta FILE ARRAY)
(geta 'infile' arr)

21. subs – find a substring containing the characters between the positions defined by two arithmetic expressions and make it the new value of a variable
example: VARIABLE := subs(VAL,EXP,EXP)
var := subs('a string',3,6)

22. trim – trim the blanks from the end of a variable value
example: (trim VARIABLE)
(trim var)

23. squeeze – replace all successive blanks within a variable value with one blank and leave a blank at the end
example: (squeeze VARIABLE)
(squeeze var)

24. fsetq – assigns a value (either a literal or the value of a variable) to a variable
example: (fsetq VARIABLE VAL)
(fsetq var 'a string)

25. fsetqa – assigns the value of an arithmetic expression to a variable
example: (fsetqa VARIABLE EXP)
(fsetqa var 3*y/2)

26. addmacsch – read definitions from a channel; stop reading when two target end-of-line flags are encountered (usually $$)
    example: (addmacsch CHANNEL)
    (addmacsch '4)

27. addmacs – read definitions from a file; stop reading when two target end-of-line flags are encountered (usually $$)
    example: (addmacs FILE)
    (addmacs 'defsfile)

28. addmacs – read definitions which immediately follow the addmacs commands; stop reading when two target end-of-line flags are encountered (usually $$)
    NOTE: this command cannot be used on channel 4 (the users terminal); instead use: (addmacsch '4)
    example: (addmacs)

29. system – perform an operating system command
    example: (system VAL)
    (system 'vi defsfile)

30. escape – temporarily escape to the operating system
    example: (escape)

31. trace – turn tracing of commands on or off
    example: (trace VAL)
    where VAL evaluates to on or off

32. fexecute – execute the value of a variable
    example: (fexecute VAR)
    (fexecute codevar)

Definition of non-keywords used in examples above:

CHANNEL	a variable or quoted literal value which evaluates to a channel number (0–9 or a–z) NOTE: channels 0–5 are reserved by the system for specific purposes and may not work with some commands
FILE	a variable or quoted literal value which evaluates to a file name
VARIABLE	a variable name which can consist of any sequence of characters which are balanced with respect to parentheses

VAL — either a variable or a quoted literal value

EXOP — can be one of four possible values:

0	rewind the file
+	do not rewind the file
0 label	rewind the file and move past a line which begins with label
+ label	do not rewind the file but move forward past a line which begins with label

EXP — an arithmetic expression which evaluates to an integer
NOTE: an arithmetic expression can involve the four arithmetic operations +, –, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)

ARRAY — an array name which can consist of any sequence of characters which are balanced with respect to parentheses

STATEMENTS — can be any sequence of zero or more statements
NOTE: each statement must fit on one line (usually 80 characters but implementation dependent) so all non–keywords have an implied limit to their size

APPENDIX 9.3

BARREL/ASP BLISP KIT

Informal description of the lisp functions available in the blisp kit. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (car #) – find the first s-expression of the non-null list
   example: (car S-EXP)
   (car '((a b c) x y z))

2. (cdr #) – find the list that is left when you remove the car of the non-null list
   example: (cdr S-EXP)
   (cdr '((a b c) x y z))

3. (cons # #) – insert the s-expression onto the front of the list
   example: (cons S-EXP S-EXP)
   (cons 'a' '(b c))

4. (eq # #) – return TRUE if the atom specified by the first argument is the same as the atom specified by the second argument; return FALSE otherwise
   example: (eq S-EXP S-EXP)
   (eq 'harry' 'harry)

5. (atom #) – return TRUE if the argument is an atom, return FALSE otherwise
   example: (atom S-EXP)
   (atom 'harry)

Definition of non-keywords used in examples above:

S-EXP	a variable or quoted literal which evaluates to an atom (literal) or a function or a list of S-EXPs

APPENDIX 9.4

BARREL/ASP BICON KIT

List of the Icon functions available in the bicon kit.

1. find
2. upto
3. any
4. many
5. move

APPENDIX 9.5

BARREL/ASP BGIGI KIT

Informal description of the commands available in the in the bgigi kit which allow access to the GIGI/Regis graphics terminal. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (open gigi) – open gigi terminal for writing graphics commands (change it from normal mode to graphics mode)
   example: (open gigi)

2. (write gigi #) – send gigi graphics command to the terminal
   example: (write gigi GRAPH)
   (write gigi p[180,50])

3. (writef gigi #) – send gigi graphics command to the terminal and write it to a file specified by the variable %gfn
   example: (writef gigi GRAPH)
   (writef gigi p[180,50])

4. (close gigi) – close gigi terminal (change it from graphics mode to normal mode)
   example: (close gigi)

5. (reset gigi) – reset all graphics attributes of the gigi terminal
   example: (reset gigi)

Definition of non-keywords used in examples above:

GRAPH a string of GIGI/Regis graphics commands

APPENDIX 9.6

BARREL/ASP BCNTRL KIT

Informal description of the while and repeat control commands available in the bcntrl kit. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. while (#) do – beginning of while loop; loop while numeric expression is true; cannot be nested

   begin – marks the beginning of the statements in the loop

   end while – marks the end of the statements in the loop

   example:
   ```
   while (BOOLOP EXP EXP) do
   begin
   STATEMENTS
   end while

   while (le a b) do
   begin
   (ty 'a is less than b)
   (fsetqa a a+1)
   end while
   ```

2. repeat – beginning of repeat until loop; cannot be nested

   until – marks the end of the statements in the loop

   (#) – loop until the numeric expression is true

   example:
   ```
   repeat
    STATEMENTS
   until
   (BOOLOP EXP EXP)

   repeat
   (ty 'a is less than b)
   ```

```
(fsetqa a a+1)
until
(ge a b)
```

3. {# – beginning of while loop; loop to corresponding label at end of loop; can be nested

   while (#) do – marks the beginning of the statements in the loop; loop while the alphanumeric expression is true; only equal and not equal are supported

   #} – marks the end of the statements in the loop; the label corresponds with the label at the beginning of the loop

   example:
```
{LIT
while (BOOLOP VAL VAL) do
STATEMENTS
LIT}
```

```
{loop1
while (ne a b) do
(ty 'a is not equal to b)
(fsetqa a a+1)
loop1}
```

Definition of non–keywords used in examples above:

BOOLOP	a boolean operator; can be any of eq, ne, gt, lt, ge, le
VAL	either a variable or a quoted literal value
EXP	an arithmetic expression which evaluates to an integer NOTE: an arithmetic expression can involve the four arithmetic operations +, –, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)
STATEMENTS	can be any sequence of zero or more statements NOTE: each statement must fit on one line (usually 80

	characters but implementation dependent) so all non-keywords have an implied limit to their size
LIT	a literal constant value

APPENDIX 9.7

BARREL/ASP BCASE KIT

Informal description of the case statement available in the bcase kit. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (casen # of) – beginning of numeric case statement

   (whenn # = > #) – individual tests of case statement; if the expression is equal to the number in the casen statement then the statement is executed

   (whenn others = > #) – default statement to execute if no matches have been found in previous whenn statements

   example:
   ```
   (casen EXP of)
   (whenn EXP = > STATEMENT)
   (whenn others = > STATEMENT)

   (casen var*3 of)
   (whenn 3 = > (ty 'var is 1))
   (whenn 6 = > (ty 'var is 2))
   (whenn others = > (ty 'others))
   ```

2. (case # of) – beginning of alphanumeric case statement

   (when # = > #) – individual tests of case statement; if the string is equal to the string in the case statement then the statement is executed

   (when others = > #) – default statement to execute if no matches have been found in previous when statements

   example:
   ```
   (case VAL of)
   (when VAL = > STATEMENT)
   (when others = > STATEMENT)
   ```

```
(case var of)
(when 'xyz' = > (ty 'its xyz))
(when 'abc' = > (ty 'its abc))
(when others = > (ty 'others))
```

Definition of non-keywords used in examples above:

VAL	either a variable or a quoted literal value
EXP	an arithmetic expression which evaluates to an integer NOTE: an arithmetic expression can involve the four arithmetic operations +, -, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)
STATEMENT	can be any one statement NOTE: each statement must fit on one line (usually 80 characters but implementation dependent) so all non-keywords have an implied limit to their size

APPENDIX 10.1

BARREL/ASP BLOGO TAILORED SYSTEM

Informal description of the logo commands available in the blogo tailored system. They are modelled after Apple Logo and are implemented in terms of the GIGI/Regis graphics terminal and commands. Required are the BGIGI definitions and the forward.r program (a C program which does the math required by Logo that ASP cannot handle). Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. right # – turn the turtle right n degrees
   example: right EXP
   right 90

2. left # – turn turtle left n degrees
   example: left EXP
   left 90

3. forward # – move turtle forward n pixels
   example: forward EXP
   forward 10

4. back # – move turtle backwards n pixels
   example: back EXP
   back 10

5. home – put turtle in center of screen and point up
   example: home

6. cs – clear the screen and go home
   example: cs

7. clean – clear the screen but don't move the turtle
   example: clean

8. penup – make the turtle pen inactive
   example: penup

9. pendown – make the turtle pen ready to draw
example: pendown

10. hideturtle – make the turtle invisible
example: hideturtle

11. showturtle – make the turtle visible
example: showturtle

12. repeat # [#] – execute a list of commands specified by the second argument (separated by underscores); repeat a number of times specified by the first argument
example: repeat EXP [STATEMENT{_...}]
repeat 5 [forward 5_right 50]

13. setbg # – set the background color
example: setbg EXP
setbg 1

14. setpc # – set the pen color
example: setpc EXP
setpc 2

15. setx # – set x coordinate
example: setx EXP
setx 90

16. sety # – set y coordinate
example: sety EXP
sety 90

A sample BLOGO program – the familiar POLYSPI Logo program (note the use of the BBAS, BGIGI, and BCNTRL statements):

```
(fsetq distance '1)
(fsetq angle '123)
(fsetq increment '3)
(open gigi)
(reset gigi)
home
while (le distance 3*angle+angle/4) do
begin
forward distance
```

```
 right angle
 (fsetqa distance distance+increment)
 end while
 (close gigi)
```

Normally, the definitional facilities of ASP are used to implement Logo procedures. So the POLYSPI program above might be defined as:

```
 polyspi # # #|
if #14 > 3*#24+#24/4 skip 3$    defined in BSYS
 forward #14$
 right #24$
 polyspi #14+#34 #24 #34$
$
```

and then called by:

```
 polyspi 1 123 3
```

Definition of non-keywords used in examples above:

EXP	an arithmetic expression which evaluates to an integer NOTE: an arithmetic expression can involve the four arithmetic operations +, -, *, / (addition, subtraction, multiplication, and division) with numbers and/or variables and/or functions as operands using balanced parenthesis as needed or desired to effect precedence (although numbers can serve as variable names such variable names cannot appear in an expression as they will be interpreted as numbers)
STATEMENT	can be any one statement NOTE: each statement must fit on one line (usually 80 characters but implementation dependent) so all non-keywords have an implied limit to their size

APPENDIX 10.2

BARREL/ASP BED TAILORED SYSTEM

Informal description of the commands available in the barrel editor. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (cd # #) – delete the first specified line and move it after the second specified line
   example: (cd LINE LINE)
   (cd 5 20)

2. (f #) – find the next line beginning with the argument
   example: (f LIT)
   (f this line)

3. (dup # # #) – duplicate the range of lines specified by the first two arguments after the line specified by the third argument
   example: (dup LINE LINE LINE)
   (dup 5 8 20)

4. (u) – undo the last deletion
   example: (u)

5. (renum) – renumber the file and write it out
   example: (renum)

6. (d) – delete the current line
   example: (d)

7. (d #) – delete the specified line
   example: (d LINE)
   (d 10)

8. (d # #) – delete the range of specified lines
   example: (d LINE LINE)
   (d 10 20)

9. (i #) – insert lines after the specified line until a null line is entered
   example: (i LINE)
   (i 10)

10. (s (#) (#)) – substitute the text specified by the second argument for the text specified by the first argument in the current line; the text can contain spaces
    example: (s (LIT) (LIT))
    (s (stuff) (with this))

11. (s # #) – same as above except the text cannot contain spaces
    example: (s LIT LIT)
    (s bad good)

12. (bye) – write the file out and exit the editor
    example: (bye)

13. (a #) – append the text to the end of the current line
    example: (a LIT)
    (a this stuff)

14. (v) – view the lines surrounding the current line
    example: (v)

15. (m) – modify the current line
    example: (m)

16. (m #) – modify the specified line
    example: (m LINE)
    (m 10)

17. (l) – list the current line
    example: (l)

18. (l #) – list the specified line
    example: (l LINE)
    (l 10)

19. (l # #) – list the range of specified lines (limited to 22)
    example: (l LINE LINE)
    (l 10 20)

20. (l a) – list lines 1 through 22
    example: (l a)

21. (p) – print current line plus the next 21
    example: (p)

22. (ed #) – edit the specified file
example: (ed FILE)
(ed 'afile)

Definition of non–keywords used in examples above:

FILE	a variable or quoted literal value which evaluates to a file name
LIT	a literal constant value
LINE	a literal constant line number

APPENDIX 10.3

BARREL/ASP BQBE TAILORED SYSTEM

Informal description of the relational database commands available in the bqbe tailored system. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (getr #) - get a relation from database in a file
   example: (getr FILE)
   (getr tables)

2. (project # #) - project the domains specified by the the second argument (separated by commas) over relation named in the first argument
   example: (project LIT LIT[,...])
   (project test make,cond)

Definition of non-keywords used in examples above:

FILE	a literal constant file name
LIT	a literal constant value

APPENDIX 10.4

BARREL/ASP BDT TAILORED SYSTEM

Informal description of the commands available in the decision table presentation processor. Keywords are lower case; non–terminals are upper case. A "general case" example is followed by a specific example.

1. dt – prompts for conditions in the table and displays actions; prompts are repeated until "no" is entered; if an prompt is answered with a value that begins with an * then actions for all of those conditions are displayed as long as the user enters a ";"

   example: dt
   car make ?
   cord
   condition ?
   good
   commission is 5%
   shop work needed is no–need
   manager ok is no–req
   we continue
   car make ?
   •
   •
   •

2. dec(#,#) – displays actions from the table based on the conditions provided as arguments; if an argument begins with an * then actions for all of those conditions are displayed as long as the user enters a ";"

   example: dec(LIT,LIT)
   dec(*LIT,*LIT)

   dec(cord,*Cond)
   commission is 5%
   shop work needed is no–need
   manager ok is no–req

```
*Cond = good
;
commission is 1%
shop work needed is 3-weeks
manager ok is no-req
*Cond = poor
;
```

Definition of non-keywords used in examples above:

LIT	a literal constant value

APPENDIX 10.5

BARREL/ASP BTINT TAILORED SYSTEM

Informal description of the decision table interpreter commands available in the btint kit. Keywords are lower case; non-terminals are upper case. A "general case" example is followed by a specific example.

1. (getb #) - get b-type decision table from the file
   example: (getb FILE)
   (getb 'adt)

   the table is of the form:

   header
   conditions
   = = = = = = = = = = = = = = =
   actions
   number of rules
   condition part of rule
   action part of rule
   •
   •
   •

   example:

   rem16h25c2a3p3;2;:4;4;2;nclassic r6
   is make 1:cord
   2:reo
   3:duesenberg
   is condition 1:bad
   2:good
   =
   commission is 1:1%
   commission is 2:5%

```
commission is          3:10%
commission is          4:variable
shop work is           1:1 week
shop work is           2:2 weeks
shop work is           3:6 weeks
shop work is           4:none
managers ok is         1:not required
managers ok is         2:required
6
1,1,
1,6,9
1,2,
2,8,10
2,1,
2,6,10
2,2,
1,5,10
3,1,
3,7,10
3,2,
2,6,10
```

2. (askb #) – interpret a b–type decision table from the file
   example: (askb FILE)
   (askb 'adt)

3. (getm #) – get an m–type decision table from the file
   example: (getm FILE)
   (getm 'adt)

   table is of the form:

   number of conditions
   conditions
   number of rules
   condition part of rule
   action part of rule
   •
   •
   •

number of actions
actions

example:

```
5
is make cord
is make reo
is make duesenberg
is condition bad
is condition good
6
10010
1,6,9
10001
2,8,10
01010
2,6,10
01001
1,5,10
00110
3,7,10
00101
2,6,10
10
commission is 1%
commission is 5%
commission is 10%
commission is variable
shop work is 1 week
shop work is 2 weeks
shop work is 6 weeks
shop work is none
managers ok is not required
managers ok is required
```

4. (askm #) – interpret an m–type decision table from the file
   example: (askm FILE)
   (askm 'adt)

Definition of non–keywords used in examples above:

FILE	a variable or quoted literal value which evaluates to a file name

GRADUATE SCHOOL
UNIVERSITY OF ALABAMA AT BIRMINGHAM
DISSERTATION APPROVAL FORM

Name of Candidate John Barrett

Major Subject Computer and Information Sciences

Title of Dissertation A Computerized Formal Methodology For Development Of Simulation Software

Dissertation Committee:

Kevin D. Reilly, Chairman

Warren D. Jones

Director of Graduate Program Warren D. Jones

Dean, UAB Graduate School

Date 9/30/91

PS-1428