AD-A078 035 UNIVERSITY OF SOUTHERN CALIFORNIA MARINA DEL REY INFO--ETC F/G 9/2
A FORMAL DEFINITION OF AMDL.(U)
OCT 79 P W ALFVIN F30602-78-C-0008
UNCLASSIFIED ISI/RR-79-78 NL

END
DATE FILMED
1-80
DDC

LEVEL II

ISI/RR-79-78
October 1979

Peter W. Alfvin

A Formal Definition of AMDL

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ISI/RR-79-78	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): A Formal Definition of AMDL	5. TYPE OF REPORT & PERIOD COVERED: Research rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Peter W. Alfvin	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0008, DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS: USC/Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90291	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 12 / 79
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: October 1979
	13. NUMBER OF PAGES: 78
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): RADC-ISCA, Griffiss Air Force Base, New York 13441	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

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

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

18. SUPPLEMENTARY NOTES

The author is now located at Xerox Corporation, El Segundo, California.

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

AMDL, computer language, denotational semantics, hardware description, ISPS

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

AMDL is an abstract form of the hardware description language ISPS. This report presents a formal definition of AMDL, using the techniques of denotational semantics as developed by Scott and Strachey. AMDL includes some nonstandard control and data structures which are easily handled by this definitional method. This report assumes familiarity with descriptive denotational semantics.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 952

ISI/RR-79-78
October 1979

Peter W. Alfvin

A Formal Definition of AMDL

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

This research was supported by the Rome Air Development Center under contracts F30602-78-C-008 and DAHC15-72-C-0308. Views and conclusions contained in this paper are the author's and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government or any person or agency connected with them.

CONTENTS

Acknowledgments

First, I would like to thank my thesis adviser, David Martin, for his continued support during the preparation of this report. He introduced me to the subject of formal semantics, provided valuable guidance while I was formulating my ideas, and carefully reviewed the thesis in its final stages. He performed each of these tasks with a diligence, patience and spirit that made working with him an enjoyable and rewarding experience.

UCLA faculty members Emily Friedman and Milos Ercegovac completed the thesis committee and I am grateful for their participation.

This report was prepared while I was a staff member at the Information Sciences Institute of the University of Southern California. Steve Crocker, under whom I worked, deserves special thanks for providing the opportunity and the encouragement to pursue my thesis topic. Discussions with Steve and other ISI members Leo Marcus and Dono VanMierop were also helpful.

This work was supported in part by the Rome Air Development Center under contracts F30602-78-C-008 and DAHC15-72-C-0308. The views expressed here are those of the author and not necessarily those of the U.S. Government.

The thesis was prepared using the substantial resources of ISI and Xerox, whose contribution I acknowledge.

ABSTRACT

AMDL is an abstract form of the hardware description language ISPS. This report presents a formal definition of AMDL, using the techniques of denotational semantics as developed by Scott and Strachey. AMDL includes some nonstandard control and data structures which are easily handled by this definitional method. This report assumes familiarity with descriptive denotational semantics.

1.0 Introduction

Since the introduction of ISP [Bell & Newell, 1971], the use of hardware description languages (HDLs) has increased dramatically. Used originally to describe at the programming level the instruction sets of digital computers, hardware descriptions are now being used in applications such as design automation, emulation, compiler generation, and program verification – to name only a few. This surge of activity has put the existing HDLs to a thorough test, with each application area making its own set of demands on the languages. One common demand, however, is for a precise definition of the HDLs themselves. → (cont on p 1473A)

Before we discuss this problem of defining a hardware description language, let us look at a very small but very common example of its use. Consider a "push" instruction on a typical computer of the form PUSH *reg*, *address*. A prose description of this instruction might read:

> "The PUSH instruction increments the register *reg* and moves the contents of the word pointed to by *address* to the word pointed to by the new value of *reg*."

While this definition might seem straightforward, there are a great number of possible difficulties. What if the registers were addressable memory locations and *address* points to *reg*? Is the original value or the incremented value of *reg* stored at *reg + 1*? Although it would be highly unlikely, what if the instruction PUSH *reg, address* were being executed out of *reg*? Would the instruction first be moved to an internal register, or would the *address* field of the instruction be modified during execution resulting in a reference to *address + 1*? And what if *reg* pointed to *reg – 1*? Would the final value of *reg* be the contents of *address* plus one or simply the incremented contents of *reg*? There are, of course, other clarifications that might need to be made concerning what constitutes a legal address, how overflow conditions are handled, and other such matters.

One could argue that such relatively minor details are not worth considering, for one should not be writing programs that depend on the machine's behavior in these unusual circumstances. This is a difficult argument in that it assumes that there could be an agreement upon what constitutes "unusual circumstances." One could also argue that there is nothing wrong with prose descriptions, and that this description simply needs to be expanded. While it is certainly true that the description could be more carefully worded, it is arguable that true precision could be obtained using this approach. Furthermore, there is no mechanism in English (or any natural language) which ensures that a definition is complete. It is sometimes not a simple matter to determine whether a prose definition has enumerated all possible situations. Finally, this kind of description is worthless to an application program requiring a computer description. If we are to reap the benefits of software such as compiler-compilers and general program verification systems,

we must provide a description more amenable to automatic processing.

In response to this need for descriptive methods, several hardware description languages have been developed. ISP has been modified into its present form ISPS [Barbacci et al., 1977] and is receiving continued attention. SMITE [TRW, 1977] was developed along similar lines and is also being heavily used. LCD [Evangelisti et al., 1976] is yet another HDL aimed at the same set of descriptive problems. Since this paper is concerned more with ISPS than with the other two languages, let us examine how it could be used to define the PUSH instruction we just examined. Consider the following segment of ISPS code:

```
MEM[reg]  : =  MEM[reg] + 1  NEXT
MEM[MEM[reg]]  : =  MEM[address]
```

We assume for simplicity here that *MEM* is the memory of the machine and that the registers are in the lowest memory addresses. The ": = " construct denotes assignment and "*[]*" denotes array reference. Upon examining this definition, we now can see that the register is incremented before the store takes place. The PUSH instruction need not have been defined this way, however. Some alternate definitions (with slightly different semantic meanings) are:

```
temp  : =  MEM[reg] + 1  NEXT
MEM[temp]  : =  MEM[address]  NEXT
MEM[reg]  : =  temp + 1
```

```
temp  : =  MEM[reg] + 1  NEXT
MEM[temp]  : =  MEM[address]  NEXT
MEM[reg]  : =  MEM[reg] + 1
```

Although each of these differs slightly in its meaning, it seems that taken individually, each is unambigous. Is our quest for clarity therefore over? Unfortunately, it is not. Consider the following, perfectly legal ISPS code segment intended to define the same PUSH instruction:

```
MEM[MEM[reg]: = MEM[reg] + 1]  : =  MEM[address]
```

This definition seems quite precise, but upon inspection one discovers a problem. Does *reg* get incremented before the reference to *address* or after? This question requires a clear understanding of the meaning of ISPS itself. Unfortunately, ISPS has been defined in English, just as the PUSH instruction was, and we are faced with equally difficult problems of ambiguity. Have we really made any progress then?

Yes, we have made progress. A HDL is a suitable mechanism for describing hardware. What is needed is a mechanism for defining the HDLs themselves and preferably one that can stand by

itself without further definition. To find such a mechanism, we look to the software world and find that this problem has already been addressed for the case of conventional programming languages.

There are several techniques used in the formal definition of programming languages. Four of the most popular methods are reviewed in [Marcotty et al., 1976]. One method not reviewed (although it was mentioned) was that of *denotational* semantics, developed at Oxford University [Milne & Strachey, 1976]. Briefly, this method provides a means of mapping programs into representative mathematic entities or *denotations*. This is done by assigning a mathematic function to each construct in the language and applying these functions to the program in question. We will examine denotational semantics more closely at a later point in the text.

But where does AMDL fit in? Recall that one of the goals of a hardware description is to allow it to be processed by various application programs. Unfortunately, languages such as ISPS, SMITE and LCD are defined in terms of character streams, as are conventional programming languages. These character streams would therefore have to be parsed by each of applications interested in processing programs in the language.

Recognizing this problem, the people working on ISPS defined a fully parenthesized format called GDB (Global Data Base) [Barbacci et al., 1977], which was essentially a parse-tree representation of ISPS programs. This format relieved the burden of parsing from the various application programs and thus was a big step forward.

The problem of formal definition of ISPS still existed, however. While the GDB format was more easily processed, it was no less ambiguous. Furthermore, some people wanted to be able to process descriptions written in either ISPS or SMITE. Since the languages were quite similar, it was hoped that a superset language could be defined into which ISPS and SMITE programs could be translated. Finally, there was a desire for an integrated programming environment for a HDL which allowed a user to edit, debug and process his description without having to deal with a collection of different programs, as was then the case.

It was out of this situation, and due to the presence of the powerful INTERLISP system [Teitelman, 1975], that AMDL (Abstract Machine Description Language) was developed. Like the GDB format, AMDL is essentially an "abstract" form of ISPS suitable for machine processing. Unlike the GDB format, however, it is also suitable for human use, particularly within the INTERLISP environment. Each of the AMDL constructs is implemented as an INTERLISP function with the full flexibility that such representation provides. There is even a mechanism available which allows users to deal in *conversational* AMDL, just as they can deal in *conversational* LISP (CLISP). This and many other unique INTERLISP features provide a quite rich user environment.

More importantly, however, it was possible to formally define AMDL. While ISPS and SMITE were under outside control and it was not possible to impose a definition on them from a distance, it was possible to provide a translation between them and AMDL. In fact, this version of AMDL is based almost solely on ISPS and translations between the two languages are relatively straightforward. In the future, it is suspected that AMDL will be expanded to include some of the features of SMITE so that it will become more of a superset of ISPS than equivalent to it.

The purpose of this report, then, is to formally define AMDL. Hopefully this will serve several purposes. First, it should ensure consistency of interpretation of AMDL among the various applications being developed. Second, it should show how denotational semantics can be used to successfully define a complete, nontrivial language. And finally, it will point out the necessity for formal definitions in general and for ISPS and SMITE in particular.

As with any formal definition, the going gets extremely thick at times. Moreover, unless one has been exposed to denotational semantics in the past, the equations will undoubtedly appear quite forbidding. The definition in this report is intended primarily for AMDL, ISPS or denotational semantics aficionados. Even then, it is unlikely that one could or would even want to read it in one sitting. As a reference for the definition of AMDL, it is hoped that it might be useful for a somewhat larger audience in resolving individual questions of interpretation.

The report is divided into five sections, the first being this introduction. Section 2 gives an introduction to AMDL and its relationship with ISPS. Section 3 discusses the notation of denotational semantics as it is used here. Section 4, the bulk of the report, provides a step-by-step presentation of the formal definition itself. Section 5 provides a summary of issues; it is followed by an appendix containing a copy of the entire definition for reference purposes.

2.0 AMDL

As mentioned earlier, AMDL is an *abstract* form of ISPS. That is, it retains the fundamental semantic properties of ISPS while providing a more convenient representation for semantic processing. The representation is more convenient primarily because it is built up from structured objects, in this case lists. This list representation is not identical to the *abstract syntax* originally introduced in [McCarthy, 1966], but lends itself nicely to the LISP environment in which most of the applications have been developed.

In describing the AMDL syntax, we will be utilizing the standard LISP output format for lists. That is, lists will be represented using enclosing parenthesis, with spaces separating the individual elements. Text will be shown in an alternative typeface, with AMDL keywords appearing in boldface. For example, (**bits** foo (**pair** 3 0)) is an AMDL construct which is a list consisting of three elements, the third of which is the construct (**pair** 3 0). The words **bits** and **pair** are AMDL keywords.

This section is divided into two parts: the first part gives a complete introduction to AMDL and assumes only that the reader is familiar with high-level language concepts in general; the second, directed at ISPS users, describes the relationship between AMDL and ISPS. For those quite familiar with ISPS, this second part might substitute for the first as an introduction to the language.

2.1 Introduction to the Language

With a few significant exceptions, AMDL is a simple ALGOL-like programming language. An AMDL *program* is essentially a parameter-less *procedure*. Procedures are composed of *declarations* and executable statements (hereafter referred to as *actions*). The declarations define *variables* and/or other procedures. The actions provide for the manipulation of values of the declared variables and changes in the flow of control (e.g., loops, procedure calls). What then, makes the language unique? It is the types of variables that are declared and the way in which the flow of control is altered that gives AMDL (and its definition) its interesting qualities.

2.1.1 Variables

AMDL programs deal almost exclusively with the manipulation of *bitstrings*. A bitstring is an indexed sequence of one or more bits. For example, to declare a bitstring variable **foo** four bits long, numbered left-to-right from 3 to 0, one would use (**bits foo (pair** 3 0)). To declare the reverse ordering of bitnames, one would exchange the arguments to the **pair** expression, as in (**bits foo1 (pair** 0 3)). Variables consisting of only one bit can be declared with a shorthand notation which omits the necessity for the **pair** expression. To declare a bitstring consisting of one bit numbered 5, one would use (**bits foo2** 5). The bit numbers (and hence the arguments

to the **pair** expression) must be nonnegative. The last element in the **bits** form is known as a *structure*, in that it describes the structure of the bitstring.

Variables which are *arrays* of bitstrings may also be declared. The mechanism and rules for numbering the elements or *words* of the array are similar to those for numbering the bits in a bitstring. For instance, to declare an array **foobar** of 15 elements numbered in ascending order beginning with 100, one would use:

(bits (words foobar **(pair** 100 114)) **(pair** 3 0))

Note that this is similar to the declaration for **foo** above, except that instead of having just **foobar** as the second element in the **bits** expression, we have **(words** foobar **(pair** 100 114)), where the third element in the **words** expression is a *structure* describing the numbering of the array. The **words** expression always occurs *inside* of the **bits** expression, if both are present. The same convention of allowing a single number to denote a pair of identical numbers is used, so that **(bits (words** foobar1 100) 0) would declare an array of one word named 100, containing one bit named 0.

To *reference* a variable, one uses almost identical forms as those used to declare a variable. For instance, to access the middle two bits of the four-bit variable **foo** discussed above, one would use **(bits** foo **(pair** 2 1)). The leftmost (high-order) bit could be accessed by **(bits** foo 3). To refer to the entire bitstring, one could use **(bits** foo **(pair** 3 0)) or just simply **foo**, which implies reference to the full variable. As might be expected, access to arrays uses the **words** form. To access the second word of the array **foobar** above, one would use **(words** foobar 101). Only one word can be referenced at a time, therefore, **pair** expressions are not allowed in the **words** form in this case. A sub-bitstring of a word can be referenced, however, as with **(bits (words** foobar 101) **(pair** 1 0)), which is referring to the two low-order bits of the second word of **foobar**.

There is another important difference between the forms used for accessing variables and declaring variables. When declaring variables, only constants are allowed in naming the words and bits. In accessing variables, however, *expressions* can be used. We will examine expressions soon, but for now it is sufficient to say that an expression is a form which is evaluated at run-time to produce a **nonnegative** *value*. This value is then used in place of a constant for indexing into the variable. Expressions can occur in two places in an access. First of all, they can be used in a **words** form to reference a word in an array, as in **(words** foobar *exp*). In the above example, *exp* must have a value lying between 100 and 114. Secondly, an expression can occur in an access to a specific bit of a bitstring, as in **(bits** foo *exp*). In this example, *exp* must have a value between 0 and 3. An expression *cannot* occur within a **pair** form in this context. For instance, **(bits** foo **(pair** *exp* 1)) would not be legal, even if *exp* always resulted in a value within the legal range. This restriction is imposed so that the *length* (in bits) of any variable

access cannot vary at run time. This third element in the **bits** access form is known as a *structure reference.*

2.1.2 Expressions

We just mentioned that an *expression* is a form that is evaluated at run-time to produce a non-negative value. Actually, it produces a bitstring, which can be interpreted in any way desired. One of these interpretations is as a base-two, nonnegative integer, producing the *value* referred to above. We also speak of the *length* of an expression in bits and will see how, as with variable references, the length of an expression remains constant at run-time. Because a value and length uniquely determines a bitstring, one can think of an expression as producing a ⟨value, length⟩ pair, rather than a bitstring, when it is convenient to do so.

2.1.2.1 Constants

A *constant* is a bitstring whose value does not change at run time. The nonnegative integers we saw earlier in variable declarations and variable accesses are actually a subclass of constant. They have an explicit *length* equal to one greater than the number of bits required to represent them in base two. For instance, the integer **2** would have a length of **3**, the integer **5** would have a length of **4**, and so on. There are three other, nearly identical, types of constants whose forms are

(hconst *val len*), **(bconst** *val len*) and **(oconst** *val len*)

where *val* and *len* are nonnegative integers giving the *value* and *length* of a constant bitstring. The reason for the three equivalent forms of constant is not easily explained in this context, but is not important for our purposes. Any of these forms, though, could appear within a *structure* or *structure reference* in place of a simple, nonnegative integer. The *length* component is then just ignored.

2.1.2.2 Variable References

We have already examined the variable reference. A variable reference is a legal expression.

2.1.2.3 Data Operator Expressions

There are a large number of data operators defined in AMDL. Most of these are binary operators and occur in expressions of the form (*operator exp1 exp2*), where *operator* is some AMDL keyword. In addition, there are a few unary operators, used in expressions of the form (*operator exp*). The data operators are further discriminated by *mode* into *unsigned* and *two's complement* operators, corresponding to the ways in which the operators interpret their arguments. The keywords for unsigned operators each begin with **us** while the keywords for two's complement

operators each begin with **tc**. The unsigned operators interpret their arguments as either boolean strings or as unsigned representations of integers.

Examples of binary unsigned operators which interpret their arguments as logical bitstrings are **usor**, **usand**, **usxor** and **useqv** which implement the basic boolean functions on a bitstring basis. Shift operators, both right and left and with various "fill" bits, are performed by **ussr0**, **ussr1**, **ussrr**, **ussrd**, **ussl0**, **ussl1**, **usslr** and **ussld**. The concatenation of two bitstrings is achieved by the **usconc** operator. In addition, there is a set of binary operators which occur in both unsigned and two's complement versions. The unsigned versions of these are as follows: **usplus**, **usdifference**, **ustimes**, **usquotient**, **usremainder**, **useql**, **usneq**, **usless**, **usgtr**, **usleq**, **usgeq**, and **ustst**. The two's complement versions have identical names, except with "**tc**" substituted for "**us**". Each of these operators has particular rules for dealing with operands of differing lengths which we will not discuss here. The rules will be given explicitly, however, in the formal definition itself.

Unary operators are **usnot**, which gives the logical complement of its operand, and **tcminus**, which gives the two's complement negation of its operand.

2.1.2.4 Substring expression

The form (**ussub** *exp structure*) is available in order to refer to a substring of an arbritary expression. In this case, the numbering of the bits in the expression *exp* is implicitly taken to be ascending, right-to-left beginning with 0. In other words, the form (**ussub** *exp* (**pair** 1 0)) will always return the right two bits of the expression; if the bit substring referenced extends beyond the length of the expression, then zero padding on the left is assumed.

2.1.2.5 Assignment

Assignment in AMDL is performed using one of two *transfer operators*. The general form of an assignment is (*transfer-op* (*var1 var2 ... varN*) *exp*), where the bitstring returned by the expression *exp* is assigned to the *concatenation* of the variables *var1 var2 ... varN*. That is, the *vari* can be viewed as destination receptacles which are lined up in sequence. The bits of the source expression *exp* are then deposited into the receptacles, one bit at a time, beginning with the rightmost bit of the source and the rightmost bit of the destination. The transfer operators **usset** and **tcset** correspond to the two modes of arithmetic interpretation. They differ in the way in which they extend the source value in the case of unequal source and destination lengths.

2.1.2.6 Procedure Calls

Procedure calls are performed with the form (**call** *variable-reference exp1 exp2 ... expN*). The expressions are the actual parameters to the procedure being called. They are evaluated from left

to right and passed *by value* to the procedure. There is no way to pass parameters by reference and no way to store values into actual parameters upon return. The number of actual parameters must match the number of formal parameters. Procedures cannot be called recursively.

The fact that the second element in the **call** form is a variable reference deserves comment. In AMDL, identifiers can denote variables, procedures, or *both*. A procedure call can be used as an expression when it refers to an identifier representing both a procedure *and* a variable. The semantics of the **call** is as follows:

1. Evaluate the parameters.
2. Call the procedure associated with the identifier contained in the variable reference.
3. After the procedure returns, evaluate the variable reference to obtain a bitstring which is the expression value.

We will see later that there is no special status for variables whose identifier is also associated with a procedure. That is, they can be accessed in the same manner as variables which do not have a procedure associated with them. Furthermore, they are not implicitly stored into upon return from the associated procedure. This would allow for some rather unusual programming practices, such as two procedures storing into each other's associated variable.

2.1.3 Actions

Actions are the next major AMDL construct. Two forms we have already examined as expressions, the assignment and procedure call, are legal actions as well. The remaining actions are described below.

2.1.3.1 Conditional Action

A simple conditional statement of the form (**cond** (*exp action*)) is provided. Its semantics are straightforward. If the result of evaluating the expression produces a nonzero value, the action is evaluated.

2.1.3.2 Sequences of actions

The form (**seq** *act1 act2 ... actN*) is used for sequencing two or more actions. The semantics is to perform the actions in order from left to right. There is also a form, (**par** *act1 act2 ... actN*), for specifying parallel execution of actions. The semantics of this form is currently identical to the semantics of the **seq** form, but will be generalized at a later date. Finally, there is a construct (**repeat** *action*) which repeats the execution of a specified action an indefinite number of times. Note that the operation of these sequencing mechanisms is subject to alteration by one of the following control actions.

2.1.3.3 Labels and control actions

An action, usually itself a sequence of actions, can be *labelled* for subsequent reference within the body of the action. A label is attached with the form (**label** *identifier action*). This label is in effect at all points within the action unless superseded by another label with the same name.

There are three control actions which can reference labels. The first is of the form (**leave** *identifier*). Execution of this action will cause control to be passed to the first action *following* the action labelled with the given name. If no such action exists within the current procedure, then the identifier must refer to a *procedure* of that name. The procedures which can be referred to are those that are statically accessible (using the same conventions as for variable access) and activated. If both of these criteria are met, then the semantics of the **leave** is to return from the specified procedure.

The second control action is of the form (**restart** *identifier*). Like the **leave** action, it refers to a label, if present. In this case, the semantics are to *restart* the execution of the action associated with the label. In case no label of the given name is present, the same rules for referencing procedures apply as for the **leave** action. However, instead of returning from the referenced procedure, the procedure is *restarted.* This restarting process can be thought of as discarding any procedure activations between the current procedure and the referenced procedure, and beginning execution at the first action in the referenced procedure.

The third control action is the **resume** action, of the form (**resume** *identifier*). This action can *only* refer to a procedure and specifically *not* to the current procedure. That is, it is a mechanism for resuming execution of a procedure other than the current one. Its semantics can be thought of in terms of the **leave** action. Performing a **resume** of a procedure is identical to performing a **leave** of the procedure which that procedure last called. For instance, if procedure A calls procedure B, and procedure B calls procedure C, then a **resume** of procedure A would have the same effect as a **leave** of procedure B. Likewise, (**resume** B) would produce the same results as (**leave** C), at this point.

2.1.3.4 Decode action

The **decode** action is similar to a *select* or *case* statement in other programming languages. It has the form

```
(decode exp
   (selector action1)
   (selector action2)
        .....
   (selector actionN))
```

where the successive selectors are compared to the value of the expression until one of them "succeeds." When that occurs, the associated action is executed. The selector can take one of the following three forms:

otherwise
structure
(*structure1 structure2 ... structureN*)

The structures referred to are like those discussed in the section on variables. That is, a structure is either a constant or a **pair** form with constant arguments. The **otherwise** selector always succeeds and would reasonably only appear as the last selector, if present at all. The second form specifies a single structure. If the value of the expression falls within the bounds specified by that structure, then the selector succeeds. (In the case where the selector is a single constant, then the value of the expression must equal that constant.) The third form is a means for specifying a list of structures such that if the value of the expression falls within the bounds specified by any of the structures, the selector succeeds.

2.1.4 Declarations

The last group of AMDL constructs are the declarations. We have already examined simple variable declarations (Section 2.1.1). The other two types of declarations are the procedure declaration and the overlay declaration.

2.1.4.1 Procedures

The form of a procedure declaration is as follows:

(**proc** *var* (*var1 var2 ... varN*) *dec1 dec2 ... decN action*)

where the "*var*"s are variable declarations like those discussed in the first section. The specific term *var* above (the second element in the form) has a dual purpose. First of all, the identifier contained within *var* is the identifier that is associated with the procedure definition. Secondly, however, if *var* specifies a structure (that is, if it is a **bits** form) then this serves as a variable declaration as well. This is the mechanism whereby an identifier can be the name of both a variable and a procedure.

The list (*var1 var2 ... varN*) specifies the formal parameters of the procedure. Each of the *vari* in the list must be a *non-array* variable declaration. When this procedure is called, the values of the actual parameters are assigned to the formal arguments.

The declarations *dec1 dec2 ... decN* are optional and specify declarations which are local to this procedure. All variables are treated *statically* for purposes of allocation. They are not initialized

at procedure entry and retain their value from one call to the next.

The action *action* in the declaration gets executed when the procedure is invoked.

2.1.4.2 Overlays

The overlay construct is a method of associating new names with previously defined variables. Its general form is (**over** *var* (*var1 var2 ... varN*)). Very briefly, the semantics of this declaration is to indicate that ensuing references to *var* are to be treated as references to the concatenation of *var1 var2 ... varN*, even in the case of an assignment. The full semantics of this declaration is quite complicated. Rather than enter into a full explanation here, we will give a few simple examples. Consider the following declarations:

```
(bits acc (pair 7 0))
(bits ext (pair 7 0))
(over (bits double (pair 15 0)) (ext acc))
```

These declarations can be viewed as establishing two registers, **acc** and **ext**, and then defining **double** as the concatenation of those two registers. For example, the following pairs of references would then be equivalent.

```
ext                                    (bits double (pair 15 8))
(bits acc (pair 2 0))                  (bits double (pair 2 0))
(usconc (bits ext 0) (bits acc 7))     (bits double (pair 8 7))
```

Another set of declarations equivalent to those above is as follows:

```
(bits double (pair 15 0))
(over (bits acc (pair 7 0)) ((bits double (pair 7 0))))
(over (bits ext (pair 7 0)) ((bits double (pair 15 8))))
```

Overlays can also be used in conjunction with arrays. We won't go into those examples here, but they are discussed thoroughly in the formal definition.

2.2 Relation to ISPS

The relation between ISPS and AMDL is very close to the relationship between a language and the domain of its parse trees. That is, the translation between ISPS and AMDL is essentially a syntactic transformation. There are some exceptions to this, however, and they will be identified in the following paragraphs.

ISPS, like most concrete languages, utilizes various special characters to delimit the various syntactic entities. Sequential actions, for instance, are separated by the keyword "NEXT" while parallel actions are separated by semicolons. Square brackets are used for array references while angle brackets are used for bit references. Labels are distinguished by the ":=" that follows them, and so on. In AMDL, there is one basic syntactic form, the list. The first element of this

list is used, if necessary, as a keyword to distinguish the construct represented by the list. For example, the ISPS phrase **foo[arg1]<arg2>** would be represented in AMDL as **(bits (words foo arg1) arg2)**. The rules for this conversion, though, are quite straightforward, as will be shown.

This section is organized roughly along the lines of the ISPS reference manual. Each ISPS language construct will be discussed in turn and related to its corresponding AMDL construct, if such a construct exists.

2.2.1 Character Set, Identifiers and Constants

All of the special characters required for syntactical purposes in ISPS are absent from AMDL. The rules for constructing identifiers are unchanged, though, except that AMDL distinguishes between upper and lower case letters.

Representation of constants is done somewhat differently in AMDL. To review, there are four different types of ISPS constant expressions: decimal, binary, octal and hexadecimal. Decimal constants are represented in AMDL as they are in ISPS, by the decimal representation of an integer. The conventions for "length" are the same; the constant is understood to be one bit longer than the number of bits needed to represent it. The difference comes in the handling of the remaining types of constants. Rather than using special characters to denote the base of the constant and then interpreting the following numeric term as a number in that base, AMDL uses the forms **(bconst** *value length*), **(oconst** *value length*) and **(hconst** *value length*) to represent binary, octal and hexadecimal constants, respectively. However, the keywords **bconst**, **oconst** and **hconst** function only as comments. In each case, the terms *value* and *length* are decimal numerals giving the unsigned value of the constant and its length in bits, respectively. The distinctive keywords are only utilized when *displaying* the constants (e.g., in a prettyprint).

There are no comments, aliases or text strings in AMDL. Name pairs, represented with an intervening colon in ISPS, are represented with the form **(pair** *exp exp*).

2.2.2 ISPS Descriptions

In ISPS, one deals with entity-heads which may or may not have a formal connection set or formal structure set. Associated with these entity-heads is an entity-body, which may be either null, a section list, a behavioral expression or a entity-formal-structure-map. There is a large number of syntactic combinations that can be made from these heads and bodies, many of which are meaningless (such as a entity-head with a formal connection set and a null body). AMDL retains the intended power of this mechanism while greatly simplifying the syntax.

There are three types of declarations in AMDL: a *procedure* declaration, a *variable* declaration and an *overlay* declaration. The variable declaration corresponds to an entity-head with a structure, but no formal connection set and a null entity-body. Replacing the square brackets,

however, is the AMDL form **words**. For instance, foo[5] would be (**words** foo 5) in AMDL. In a similar fashion, the **bits** form replaces the angle brackets. The ISPS term foo<7> would be (**bits** foo 7) in AMDL. The combined case of foo[5]<7> would be represented as (**bits** (**words** foo 5) 7), with the **bits** form always surrounding the **words** form. The use of name pairs in this context is straightforward. For instance, foo[5]<7:0> would be (**bits** (**words** foo 5) (**pair** 7 0)).

The procedure declaration provides the context for the other types of declarations. Its syntax, which is similar to that in most programming languages, is:

(**proc** *var* (*var1 var2 ... varN*) *dec1 dec2 ... decN action*)

The second element of the list corresponds to the ISPS entity-head. The third element is the (optional) list of formal parameters. Following that are zero or more local declarations and finally the required action for the procedure. The formal parameters are treated as local variable declarations and may not be array variables. Upon procedure activation, parameters are always passed *by value*.

This procedure declaration mechanism provides the ability to utilize a single identifier for representing a variable and a procedure. This ability corresponds to the ISPS feature of allowing separate specification of structure and behavior. In a manner similar to ISPS, if the *var* component of the declaration is just an identifier, then no variable declaration takes place (the entity has no *structure*, in ISPS terms). A call to this procedure could never be used in an expression in that it wouldn't "return" a value. However, if *var* is a variable form (i.e., its keyword is **bits**), then it is treated as a declaration. As in ISPS, the identifier has the status of a bona fide variable and can then be accessed independently from its associate procedure.

The AMDL overlay corresponds to the ISPS entity-head with a E-FS-Map. It is a means of mapping a new variable onto one or more previously defined variables. The syntax is (**over** *var* (*var1 var2 ... varN*)), where the *var*s are subject to rules similar to those for ISPS in the same situation.

2.2.3 Behavior Expressions

The corollary to the ISPS behavioral expression is simply the AMDL procedure action. AMDL retains the word *action*, coined by ISPS, in place of statement, but there need be no distinction. The ISPS description of an action as "... the sequence of transformations and transfer of values stored in carriers ..." could as easily be applied to an executable statement occurring in any programming language. It involves variables, constants, loops, procedure calls and other such common things.

The mapping from ISPS actions to AMDL actions is again purely syntactic. Sequential actions

are represented by (**seq** *act1 act2 ... actN*). Parallel actions use the same form except with **par** substituted for **seq**. There are no "named blocks" in AMDL. An "if" statement is represented by (**cond** (*exp act*)). The control operations take the form (**repeat** *act*), (**leave** *id*), (**restart** *id*) and (**resume** *id*). Procedure calls are accomplished by the form (**call** *id exp1 exp2 ... expN*), where *id* is the name of the procedure and the *exp*s are the actual parameters to the procedure.

The decode statement is derived directly from the ISPS syntax. The general format is:

```
(decode exp
   (selector action1)
   (selector action2)
        .............
   (selector actionN))
```

where *selector* is either the keyword **otherwise**, a name pair, a constant, or a list of name pairs and constants. The semantics of these selectors is identical to that of ISPS.

2.2.4 Carrier Expressions

ISPS provides a hierarchical definition of carrier expressions which implies a specific precedence for expression evaluation. No such hierarchy is required in AMDL, where all such operations are specified in prefix form. Operations are either *binary* operations or *unary* operations, with the forms (*binary-op exp1 exp2*) and (*unary-op exp*), respectively.

A subset of the operations provided in ISPS are provided in AMDL. The one's complement and signed magnitude modes of arithmetic have been omitted, leaving only unsigned and two's complement. Furthermore, qualifiers are not supported for the purposes of specifying the current mode. All operations carry a specific arithmetic in their name. Specifically, all two's complement operations begin with the two letters **tc**, and all unsigned operations with the letters **us**. Those operations which behave identically in all ISPS modes contain the **us** prefix by default. A table of the ISPS operations and their AMDL counterparts is given below.

ISPS Operator	AMDL Keyword(s)
+	**usplus, tcplus**
– (binary)	**usdifference, tcdifference**
*	**ustimes, tctimes**
/	**usquotient, tcquotient**
MOD	**usmod, tcmod**
eql	**useql, tceql**
neq	**usneq, tcneq**
lss	**uslss, tclss**

leq	**usleq, tcleq**
gtr	**usgtr, tcgtr**
geq	**usgeq, tcgeq**
sr0	**ussr0**
sr1	**ussr1**
srr	**ussrr**
srd	**ussrd**
sl0	**ussl0**
sl1	**ussl1**
slr	**usslr**
sld	**ussld**
or	**usor**
and	**usand**
eqv	**useqv**
xor	**usxor**
@	**usconc**
not	**usnot**
- (unary)	**tcminus**

In ISPS, there is both a regular transfer operation and a sign-extend transfer operation, the latter being available in the various arithmetic modes. In keeping with the decision regarding data operations, AMDL supports two's complement and unsigned sign-extended transfer operations. However, it was noticed that the regular transfer operation was identical to the unsigned sign-extended transfer operation. So the need for the special form for a regular transfer operation was lost. The form of a transfer is then (*transfer-op* (*var1 var2 ... varN*) *exp*), where *transfer-op* is either **usset** or **tcset**. The list of destination variables (*var1 var2 ... varN*) would correspond to the term *var1@var2@...@varN* in ISPS.

An ISPS carrier access and/or activation corresponds to an AMDL variable access or procedure call. To access an AMDL variable, one simply uses the standard **bits** and **words** notation. To call a procedure, one uses a form similar to procedure call form used in an action, namely (**call** *var-ref exp1 exp2 ... expN*). The difference is that instead of a simple identifier as the second element in the form, a variable reference appears. This provides the mechanism of joint access and activation supported in ISPS.

2.2.5 Qualifiers and Identifier Sequences

There are no qualifiers or identifier sequences in AMDL.

2.2.6 ISPS Definitions

There are no macro. define or require statements in AMDL.

2.2.7 Predeclared Entities

There are no predeclared entities in AMDL.

3.0 Notation

As mentioned earlier, this report assumes that the reader is familiar with the descriptive techniques of denotational semantics. For those who are not, several discussions of the technique of denotational semantics can be found in the literature. Of these, [Gordon, 78] gives an exceptionally clear introduction to the subject. This section merely reviews the notational conventions used in the report. Whenever possible, the conventions used are those of Milne and Strachey as found in [Milne & Strachey, 1976]. It was felt that, despite its shortcomings, it was better to use the standard convention than introduce a variant.

3.1 Syntax

The AMDL syntax is specified by a list of syntactic domains and a set of BNF-like equations giving the definition of those domains. The list of domains is of the following form:

Δ:Dec Declarations

which states that Δ will be used to specify an element of the domain **Dec** which is the domain of AMDL Declarations. Capital Greek letters (such as Δ) are always used to represent an element from a syntactic domain. An attempt was made to match the Greek letters with the domains in a meaningful way, but this was not always possible. The use of single Greek letters to denote syntactic elements is one of the conventions of denotational semantics which may be worth re-examining.

An example should serve to explain the meaning of the syntactic equations. Again, considering the domain **Dec**, we have:

$$\Delta ::= (\textbf{proc}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_{n\geq 0})\ \Delta_1\ \Delta_2\ \ldots\ \Delta_{m\geq 0}\ A)\ |\ (\textbf{over}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n))\ |\ \Pi$$

The notation "::=" is used to define the set represented by the element on the left by the expression on the right. The vertical bar "|" is used as an alternation symbol. From this example then, we see that a declaration Δ can be any of the three forms (**proc** ...), (**over** ...) or Π. Boldface type is used to denote AMDL keywords, such as **proc** or **over**. The parentheses can be viewed in either of two ways. Since AMDL programs are actually list-structures, the parenthesis can be thought of as denoting that list structure. For instance, (**over** Π_0 (Π_1 Π_2 ... Π_n)) is a list containing three elements, the last of which is a list as well. Alternately, the syntax can be thought of as representing the way in which an AMDL program would be printed. In this case, the parentheses would be viewed as (reserved) symbols of the language. In LISP terms, it is the question of whether to model the lists themselves or their *print-names*. The distinction does not affect the semantic definition.

Subscripts are used to distinguish between individual occurrences of an element from the same

syntactic domain. Ellipses are used, as in (**over** Π_0 (Π_1 Π_2 ... Π_n)), to represent a sequence of items. The members of a sequence are indexed from 1 and, unless otherwise noted, sequences contain at least one element. Note that in the above example, Π_0 is not an element in a sequence. A more complicated example is given by (**proc** Π_0 (Π_1 Π_2 ... $\Pi_{n\geq 0}$) Δ_1 Δ_2 ... $\Delta_{m\geq 0}$ A). Here, there are two independent sequences, both of which could be empty.

The use of ellipses in denotational semantics is not limited to syntax alone; it occurs in the semantic equations as well. As part of a definitional technique whose forte is its mathematical precision, these ellipses appear at first to be somewhat out of place. Most formal definitions make use of recursion to represent such iteration. However, it is not the notation that gives denotational semantics its power, but the mathematical objects (domains, functions, etc.) being denoted. The use of ellipses is sufficient as a notational mechanism in that it is unambigous. Indeed, many people find the ellipses to be a much more natural means of representation than the recursion mechanisms which they replace.

3.2 Semantics

The denotational semantics of a programming language is defined by means of functions from the set of programs in that language into a set of denotations. In the case of AMDL, we will be defining a function P which maps elements from Pro (the set of programs) into elements from ANS (the set of answers), otherwise written as P:Pro→ANS. The function P is known as the *semantic function* for the syntactic domain Pro and is specified by a *semantic equation*. The result obtained from applying P to an AMDL program is known as the *denotation* of that program.

Before we can define the semantic functions, however, we must introduce some *semantic domains*. These domains define the types of objects with which the semantic functions will be dealing. A simple example of a pair of semantic domain definitions is:

v:V = {0,1}	Bits
β:B = V^+	Bit Strings

This says first that V is the domain of "bits," modeled by the set consisting of the integers 0 and 1. Furthermore, elements from V will be represented by variables of the form v, v_1, v_2 etc. Then a new domain B is defined in terms of V. The notation V^+ is used to represent the set of all finite, nonempty sequences of elements from V.

We also will need to introduce some operations on these domains. Since this report is concerned with *descriptive* semantics, we will not provide rigorous definitions for these operations. It is assumed, however, that the domains upon which these function operate are chain-complete, partially-ordered sets, or CPO's, and that furthermore, these functions are continous over CPO's. A more thorough treatment of this subject can be found in [Milne & Strachey, 1976], which draws the initial work of Scott [Scott, 1976].

Sequences

$\langle d_1,d_2,....,d_n\rangle$	A list containing the elements $d_1,d_2,....,d_n$
$\#d$	Number of elements in d (ex. $\#\langle d_1,d_2,....,d_n\rangle = n$)
$d{\downarrow}i$	The ith element of d (ex. $\langle d_1,d_2,....,d_n\rangle{\downarrow}i = d_i$)
$d{\dagger}i$	The ith tail of d (ex. $\langle d_1,d_2,....,d_n\rangle{\dagger}i = \langle d_{i+1},d_{i+2},....,d_n\rangle$, $0\leq i\leq n$)
$d\S e$	The concatenation of d and e (ex. $\langle d{\downarrow}i\rangle \S d{\dagger}i = d{\dagger}(i-1)$, $1\leq i\leq \#d$)

Function Spaces

$$D_1 \rightarrow D_2 = \{f \mid f{:}D_1 \rightarrow D_2 \text{ and } f \text{ is continuous}\}$$

Products

$$D_1 \times D_2 \times \ldots \times D_n = \{\langle d_1,d_2,....,d_n\rangle \mid d_i \in D_i,\ i = 1,2,3,....,n\}$$

Sequences

$$D^+ = \{\langle d_1,d_2,....,d_n\rangle \mid d_i \in D,\ i = 1,2,...\}$$

$$D^* = \{\langle d_1,d_2,....,d_n\rangle \mid d_i \in D,\ i = 0,1,2,...\}$$

Sums

$$D_1 + D_2 + \ldots + D_n = \{\langle d_i,i\rangle \mid i = 1,2,....,n \text{ and } d_i \in D_i\} \cup \{\perp\}$$

$$d \in D_i = \begin{cases} \text{true} & \text{if } d = \langle d_i,i\rangle \\ \text{false} & \text{if } d = \langle d_i,j\rangle \text{ and } j \neq i \\ \perp & \text{if } d = \perp \end{cases}$$

$$d|D_i = \begin{cases} d_i & \text{if } d = \langle d_i,i\rangle \\ \perp & \text{otherwise} \end{cases}$$

$$d_i \text{ in } D = \langle d_i,i\rangle$$

Notes:

1. The symbol $\in$ introduced above is identical in design to the standard set membership ϵ. It can only be distinguished by its size. In the semantic equations, $\in$ is used almost exclusively.

2. It is conventional to let context denote whether an element is in D_i or in $D = D_1 + D_2 + \ldots + D_n$. Specifically:

   (a) If $d_i \in D_i$ occurs in a context requiring a member of $D = D_1 + D_2 + \ldots + D_n$, then "$d_i$" should be interpreted as "d_i in D."

   (b) If $d \in D_1 + D_2 + \ldots + D_n$ occurs in a context requiring a member of D_i, then "d" should be interpreted as "$d|D_i$."

The semantic functions will be defined by first giving their associated function space, as in the following example:

A:Act→U→G→C→C

which says that the semantic function A is a member of the function space Act→U→G→C→C. Semantic functions are given by one or two letters in the italic type shown. The notation Act→U→G→C→C is short for Act→(U→(G→(C→C))). In fact, this specific domain could have been defined (though not identically) as A:(Act×U×G×C)→C. However, the practice of utilizing the "cascading" function space (known as *currying*) has advantages in terms of a shortened notation.

After the semantic domains have been given for the functions, a set of equations will be given. In general, one equation will be given for each syntactic case. Conventions used within these equations are as follows:

$\lambda\alpha\beta\gamma.E$ is equivalent to $\lambda\alpha.(\lambda\beta.(\lambda\gamma.E))$ for any E

$F\alpha\beta\gamma$ is equivalent to $((F(\alpha))(\beta))(\gamma)$

$F\alpha\beta = E$ is equivalent to $F = \lambda\alpha\beta.E$

$\alpha \circ \beta \circ \gamma = \gamma(\beta(\alpha))$

$\alpha \rightarrow E_1, \beta \rightarrow E_2, E_3$ means "If α then E_1; else if β then E_2; else E_3"

$F[\alpha/i] = \lambda n.(n = i) \rightarrow \alpha, F(n)$

Bold square brackets **[]** will surround any syntactic element used as an argument to functions.

During the definition of the semantic function, it will become necessary to introduce some auxiliary or *support* functions. These functions will always be represented using italicized, boldface type as in ***support***. These functions will be introduced as they are needed, but are listed again in the appendix for reference.

4.0 Formal Definition of AMDL

This section presents a complete definition of AMDL using the techniques of descriptive denotational semantics. First, the syntax of the language is defined through a set of BNF-like productions. Second, a set of semantic domains are introduced which form the framework around which the definition will be built. Finally, the semantic equations associated with each syntactic production in the language are given. Together these components serve to totally define the meaning of any AMDL program.

4.1 Syntax

In Section 2, we informally examined AMDL and its syntax. We used metaexpressions such as **(word** *id exp*) where the terms *id* and *exp* were loosely associated with the domains of identifiers and expressions, respectively. Then in Section 3, we introduced a slightly more formal notation. This second notation will be used from now on out. Some slight differences in organization from the earlier section may be noted, but the definitions are equivalent. The complete definition follows and is reproduced in the appendix.

4.1.1 Syntactic Domains

A:Act	Actions
B:Bin	Binary Operators
X:Con	Constants
Z:Dcl	Decode Clauses
Δ:Dec	Declarations
E:Exp	Expressions
I:Ide	Identifiers
N:Num	Numerals
P:Ref	Variable References
K:SRf	Structure References
Σ:Str	Structures
T:Tra	Transfer Operators
Υ:Una	Unary Operators
Π:Var	Variables

4.1.2 Syntactic Equations

A ::= **(cond** (E A)) | **(decode** E Z_1 Z_2 ... Z_n) | **(label** I A) | **(leave** I) |
(restart I) | **(resume** I) | **(repeat** A) | **(seq** A_1 A_2 ... A_n) |
(par A_1 A_2 ... A_n) | **(call** I E_1 E_2 ... $E_{n\geq 0}$) | E | **(write** E)

B ::= **usplus | usdifference | ustimes | usquotient | usremainder |**
useql | usneq | uslss | usgtr | usleq | usgeq | ustst |
tcplus | tcdifference | tctimes | tcquotient | tcremainder |
tceql | tcneq | tclss | tcgtr | tcleq | tcgeq | tctst |

ussr0 | ussr1 | ussrr | ussrd | ussl0 | ussl1 | usslr | ussld |
usor | usxor | usand | useqv |
usconc

X ::= **(bconst** N_1 N_2**)** | **(hconst** N_1 N_2**)** | **(oconst** N_1 N_2**)** | N

Z ::= $((\Sigma_1\ \Sigma_2\ \ldots\ \Sigma_n)$ A) | $(\Sigma$ A) | A | **(otherwise** A**)**

Δ ::= **(proc** Π_0 $(\Pi_1\ \Pi_2\ \ldots\ \Pi_{n\geq 0})$ $\Delta_1\ \Delta_2\ \ldots\ \Delta_{m\geq 0}$ A**)** | **(over** Π_0 $(\Pi_1\ \Pi_2\ \ldots\ \Pi_n)$**)** | Π

E ::= **(ussub** E K**)** | (B E_1 E_2) | (Υ E) | **(call** P E_1 E_2 ... $E_{n\geq 0}$**)** | (T $(P_1\ P_2\ \ldots\ P_n)$ E)) | P

I ::= The set of legal identifiers

N ::= The set of numerals consisting of the digits 0-9

P ::= **(words (bits** I K**)** E**)** | **(words** I E**)** | **(bits** I K**)** | I

K ::= **(pair** X_1 X_2**)** | E

Σ ::= **(pair** X_1 X_2**)** | X

T ::= **usset | tcset**

Υ ::= **tcminus | usnot**

Π ::= **(words (bits** I Σ_1**)** Σ_2**)** | **(words** I Σ**)** | **(bits** I Σ**)** | I

4.2 Semantic Domains

4.2.1 Datatypes

The only datatype in AMDL is the bitstring. There are no integers or booleans as they occur in other programming languages. The definitions relating to bitstrings are as follows:

$v:V = \{0,1\}$	Bits
$\beta:B = V^+$	Bit Strings
$e:E = \{\langle m,n\rangle \mid m,n\in N,\ n\geq 0,\ 0\leq m\leq 2^n-1\}$	Expression Values

The first two domains are straightforward. but the third might be somewhat puzzling. The elements of E are simply other (equivalent) representations for bitstrings. This representation turns out to be more convenient in several places in the definition. An element from E can be thought of as a <*value.length*> pair. where *value* is the base-two integer value of the bitstring that is being represented and *length* is the number of bits in the bitstring.

There are some additional datatypes which are used during the course of the definition:

$t:T = \{true,false\}$	Booleans
$m,n:N = \{0,1,2,...\}$	Non-negative Integers
$z:Z = \{...,-2,-1,0,1,2,...\}$	Integers
$p:P = N\times N$	Pairs of N
$\varnothing = \{\oplus\}$	Set consisting of the "undeclared" element

ERR = N	Error codes
OUT = N	Output from **write** action
ANS = (ERR + OUT)*	Final answers

The domain ∅ is used primarily in constructing the target domain of functions which need to return a "undeclared" or "undefined" value. The last three lines describe the domain ANS which models the meaning of an AMDL program as a sequence of output and error codes. It will be seen later that if an error code appears in one of these sequences, it will be the last element in the sequence (corresponding to a program error abort).

4.2.2 Stores

A *store* is a function used to model some type of memory. As is commonly done, we will represent our store as a mapping from a set of *locations* into a set of *storable values*. In order to not commit ourselves to any particular location scheme, we will simply define the set of locations L as "any countably infinite set." If we had chosen some finite set, then we would have had to address the problem of running out of locations. We could have let L be the set of nonnegative integers, but there was no reason to be that specific.

Most storable values in present-day computers are words of a fixed length. For reasons which will become evident as we proceed, we will use *bits* rather than words as our storable values. We will also find it useful to include the set F of *function abstractions* and the set ∅ in our set of storable values. F will be described later. Our domain for stores, then, looks like:

σ:S = L→(V + F + ∅)	Stores

4.2.3 Variable Descriptors

It was mentioned earlier that the primary AMDL datatype is the bitstring. But notice that we have not included bitstrings in our set of storable values. In order that we may work with stored bitstrings, we introduce the following two domains:

ω:W = L+	Word Location Strings
α:A = W+	Array Location Strings

Word location strings are simply a nonempty sequence of locations, and array location strings are simply a sequence of word location strings.

AMDL variables, however, come in varying lengths and indexing schemes (e.g., **(bits (words foo (pair** 0 15)) **(pair** 7 4)) or foo[0:15]<4>). Our description for variables must include this additional information. The following domains are introduced for this purpose:

w:WORD = W×P	Word Variable Descriptors
a:ARRAY = A×P×P	Array Variable Descriptors

The P component of a WORD is a pair of numbers which give the left and right *bitnames* of the

word variable being described. The ARRAY domain requires two P components. The first describes the bitnames, as in a WORD, but the second is needed to describe the *wordnames* of the array. For example, if "a" is an array variable descriptor being used to describe the variable foo declared by

(bits (words foo (pair 0 15)) (pair 7 4))

then a of the form

$a = \langle \alpha, \langle 7,4 \rangle, \langle 0,15 \rangle \rangle$

for some α. For example, then, the expression $a{\downarrow}3{\downarrow}2$ (= 15) can be used to access the index of the rightmost word.

4.2.4 Continuations

The theory of continuations is fundamental to descriptive denotational semantics and to this definition in particular. Anyone who is not comfortable with the idea of continuations after reading these few paragraphs is strongly urged to consult one of the references. The AMDL definition contains several types of continuations. Basically, a continuation is employed in the semantic definition of any AMDL construct whose evalutation might result in a change in the sequential flow of control, due to the possibility of errors, branches or similar occurrences. As it turns out, most of the AMDL constructs fall into this category, including the three major ones: declarations, actions and expressions.

Continuations are functions from intermediate results to final answers. Intermediate results can be thought of as the results produced by an individual construct. For instance, the intermediate result of a constant is simply the value of that constant. The intermediate result of an assignment action is a new store. The intermediate result of an assignment expression (an assignment used as an expression) is both a new store and a value. The continuations for the various types of constructs, therefore, are functions whose domains are the various types of intermediate results.

But why have continuations at all? Why not let the denotation of a construct simply be its intermediate result? To see the answer to this, consider the assignment action. As mentioned earlier, an assignment action produces a new store from an old store. One might suggest, then, that the domain for action semantics should be that of store-to-store functions. But what about the branching actions (e.g., **leave**, **restart**, **resume**). These seem to leave the store unchanged, but their denotation would certainly not be the identity function on stores! We need to convey, somehow, that these branching actions change the normal, sequential course of events. This problem is solved by letting the denotation of an action be a function of the *rest of the program*, where *rest of the program* refers to all of the remaining computation. More precisely, we let the denotation of an action be a function of an action continuation whose domain is given by

θ:C = S→ANS Action Continuations

If an action does not cause a branch, it simply applies its continuation argument to the new store that it produced. If a branch is desired, some other function (say a jump j) is applied instead.

Expressions, as mentioned earlier, produce a value in addition to a new store. But they, too, may alter the flow of control. Division by zero or procedure calls which take abnormal returns are two ways in which this may happen. The domain of expression continuations is defined as follows:

κ:K = (E×S)→ANS

That is, expression continuations are functions of expression values and stores. But by currying the above domain, we get

κ:K = E→S→ANS

which further reduces to

κ:K = E→C Expression Continuations

This type of simplification will be performed frequently throughout the definition, usually without noting the intermediate steps.

The next type of continuation is for structure references. A structure reference is that component of a variable reference that references specific bits. A continuation is needed because a structure reference may be an expression, and an expression requires a continuation. The intermediate result of a structure reference, though, is a number pair p and (due to the presence of an expression within the structure reference) a new store. This leads to

μ:M = P→C Structure Reference Continuations

But how can this continuation be used as an expression continuation? Quite easily, as will be shown later.

Variable references also require their own continuation. The intermediate result of a variable reference is a word location string and a new store. A variable reference must produce a word location string, rather than simply a bitstring, because variable references occur in the destination component of assignment statements as well as in expressions. This gives us:

η:H = W→C Variable Reference Continuations

The rest of the continuations are somewhat more complicated, involve some domains which have not yet been defined, and will be covered in the corresponding section on the semantic equations. They are listed here for completeness:

$o:O = A \rightarrow ANS$ Overlay Continuations

$\chi:X = Q \rightarrow U \rightarrow C$ Declaration Continuations

$\gamma:G = L \rightarrow ((J \times (J + \varnothing) \times (J + \varnothing)) + \varnothing)$ Procedure Abstraction Continuations

4.2.5 Other Domains

Before we look at the *environment* domain, there are few other domains that deserve attention. One of the interesting results of the theory of continuations is that the semantics of a program "jump", rather than providing great difficulties, becomes quite simple. In this case, a "jump" is simply an action continuation. This makes sense, in that a jump merely directs the computation to a different *rest of the program.* So, we have:

$j:J = C$ Jumps

We have not said anything yet about how we are intending to model the meaning of an AMDL procedure. We introduce the notion of a procedure abstraction as follows:

$f:F = E^* \rightarrow G \rightarrow C$ Procedure Abstractions

This models a procedure as a function of a sequence of expression values (corresponding to the arguments of the procedure), a procedure abstraction continuation G (to be discussed later) and a store (embedded in C).

In order that we may have a way of determining which variables have been bound at a particular textual point in an AMDL program, we introduce the domain

$q:Q = Ide \rightarrow (\{true\} + \varnothing)$ Local Binding Functions

which we will use for this purpose.

We also introduce the domain D which will be a "universal" domain to be used by support functions whose arguments are, in a sense, type less.

$\delta:D$ = All finite domains which can be constructed from given primitive domains and a finite number of construction operations.

We will also have occasion to use δ as a variable in places where it is necessary to represent elements from domains not explicitly mentioned.

4.2.6 Environment

An environment is used to associate semantic information with program identifiers. In the case of AMDL, this information deals with variables, procedures and labels. If an identifier can be used as a variable, the environment associates a variable descriptor (either word or array) with that identifier. Different environments may be in force at different times in the program. Therefore, the semantic information associated with a given identifier depends upon the environment in

whose range it lies. If an identifier can be used as a procedure, then the environment tells how many parameters that procedure must have and provides a pointer to a procedure abstraction. (The reason for using a pointer will be seen later.) Finally, if an identifier is used as a label, then the environment contains jumps (members of J) which are to be used during branches to that label. There is a fourth component to the environment, which is used to contain the unique location associated with the procedure in which the environment is being used. This fourth component can be ignored for the time being. Our domain, then looks like

$$\rho : U = UVAR \times UPROC \times ULAB \times L$$
$$UVAR = (Ide \rightarrow (ARRAY + WORD + \varnothing))$$
$$UPROC = (Ide \rightarrow ((L \times N) + \varnothing))$$
$$ULAB = (Ide \rightarrow ((J \times J) + \varnothing))$$

The reader should try to familiarize himself with the above environment domain, as it will be used frequently throughout the definition. If this is too difficult, simply remember that the first component deals with variable definition, the second with procedure definition, and the third with labels. The context in which the components are used should provide some additional help.

4.3 Semantic Functions

We are now ready to look at the heart of the definition, the semantic functions. For each construct, we will provide a semantic domain and a set of semantic equations which will collectively define the construct's semantics. The domains and equations will be discussed in detail and related to actual programming situations. In some cases, a number of alternative semantic interpretations and their denotations will be discussed.

The constructs will be presented in an order that will facilitate a bottom-up assimilation of the definition. There will be some forward references, but these will be clearly noted when present. The unary and binary operators, due to their number, will be presented last. Several "support" functions will appear throughout the semantic equations as they are required.

4.3.1 Identifiers

There is no semantic function for identifiers. Elements from **Ide** are used directly in the semantic equations.

4.3.2 Numbers (N:Num→N)

The semantic function N maps elements from **Num** into their decimal interpretation. This function is not defined by any equation.

4.3.3 Constants (C:Con→E)

The semantic function C maps constants into expression values. The first three types of constants explicitly specify their value and length, so C is just given by

$$C[(\textbf{bconst}\ N_1\ N_2)] = \langle N[N_1], N[N_2]\rangle$$
$$C[(\textbf{hconst}\ N_1\ N_2)] = \langle N[N_1], N[N_2]\rangle$$
$$C[(\textbf{oconst}\ N_1\ N_2)] = \langle N[N_1], N[N_2]\rangle$$

The other type of constant is simply a numeral N, whose meaning is an expression consisting of a value $N[N]$ and a length equal to one greater than the number of bits required to represent N. Some question arises as to the length when $N[N] = 0$. In some sense, it takes one bit to represent "0", because *no* bits have *no* meaning. On the other hand, it could be argued that it should take one more bit to represent a "1" than a "0", for the same reason that it takes one more bit to represent (say) a "4" than a "2". We have chosen the latter alternative and defined the length of "0" to be one (that is, one more than zero). So, the definition of C is completed by

$$C[N] = \langle N[N], \textbf{intlog}(N[N]) + 2\rangle$$
$$\text{for } N[N] \geq 1$$
$$C[N] = \langle 0,1\rangle$$

for $N[\![N]\!] = 0$

where ***intlog***(n) gives the integer part of $\log_2$ n.

4.3.4 Structure References (K:SRf→U→G→M→C)

A structure reference is either a pair of constants, as in (**pair** X_1 X_2) or an expression E. This is the mechanism used to refer to a particular bit field in a variable reference. Because of the possible occurrence of an arbitrary expression inside of a structure reference, the semantic function K must be a function of the environment ρ, a procedure continuation γ, a structure reference continuation μ, and a store σ. (Again, note that K could also be viewed as a member of [(SRf×U×G×M×S)→ANS], but that the above notation lends itself to shorter definitions.) Not all of these arguments are used in the case of the pair of constants, whose semantic equation is

(K1) $K[\![(\textbf{pair}\ X_1\ X_2)]\!]\rho\gamma\mu = \mu(C[\![X_1]\!]\downarrow 1, C[\![X_2]\!]\downarrow 1)$

That is, the structure reference continuation is applied to the *value* portions of the two constants, and the other arguments are ignored. The equation for the case of the expression shows how one type of continuation is utilized in creating another type of continuation. Here, we have

(K2) $K[\![E]\!]\rho\gamma\mu = E[\![E]\!]\rho\gamma\{\lambda e.\mu(e\downarrow 1.e\downarrow 1)\}$

This says that the meaning of a structure reference E is the meaning of $E[\![E]\!]$ with the same environment and procedure continuation, but with a new expression continuation. This expression continuation, $\{\lambda e.\mu(e\downarrow 1.e\downarrow 1)\}$, takes the expression value and uses its value component as both arguments to the structure reference continuation. In other words, when a single expression is used as a structure reference it is referring to a single bit.

4.3.5 Variable References (R:Ref→U→G→H→C, RI:Ref→Ide)

There are two semantic functions listed above. The second function RI will be necessary later on. It simply returns the identifier contained within the construct.

(RI1) $RI[\![(\textbf{words}\ (\textbf{bits}\ I\ K)\ E)]\!] = I$
(RI2) $RI[\![(\textbf{words}\ I\ E)]\!] = I$
(RI3) $RI[\![(\textbf{bits}\ I\ K)]\!] = I$
(RI4) $RI[\![I]\!] = I$

The other function, R, is the function which allows access to variables. As discussed earlier, its *intermediate result* is a location string. There are several things, though, which must be true for a reference to be valid:

- The variable must be defined (i.e., in the **environment).**
- The variable must be accessed in the right mode (words as words, arrays as arrays).

- The bit or word references must be within bounds.

If any of these conditions is not satisfied, then the program is considered to be in error. How, though, is an error condition represented in the conditions? First, recall that the semantic function *PROG* is a function from programs into answers, where answers are modeled by the domain ANS = (OUT + ERR)*, with the domains OUT and ERR being equivalent to the domain of nonnegative integers, N. That is to say, an answer is essentially a sequence of zero or more nonnegative integers, tagged as to whether they are from the domain OUT or from the domain ERR. The elements from OUT are generated by evaluation of **write** statements. The elements from ERR, however, are generated when a run-time error occurs via functions such as ***err***, whose definition follows:

(SF7) ***err***:N→S→ANS Program Error
err(n)σ = ⟨n in ERR⟩

Given an argument N (used to denote an error number) and a store σ, it produces a sequence consisting of the single element "n in ERR". A list of error codes and their meanings can be found in the appendix. Although the store is not used in the definition, it is present because the ***err*** function is frequently used in places where it is desirable to return a function of type S→ANS or C. It might be more appropriate to think of the definition of ***err*** as

err:N→C
err(n) = (λσ.⟨n in ERR⟩)

We return now to our discussion of the semantics of variable references. Taking the simplest case first, we have

(R4) $R[\![I]\!]\rho\gamma\eta =$
$\delta \notin \text{WORD} \rightarrow \textit{err}(11),$
$\eta(\delta\downarrow 1)$
where $\delta = \rho\downarrow 1[\![I]\!]$

which looks at the variable portion of the environment applied to the identifier I and calls it δ. If δ is undefined or is an ARRAY, then an error has occurred. Otherwise, the variable reference continuation η is applied to the entire location string δ↓1.

The second case we will look at involves specific bit references. Here, we must first determine that the bit reference is valid and then reference the appropriate bits. Determining the validity of the bit reference is done via the support function ***legal***

(SF10) ***legal***:(P×P)→T Determines if access is legal
legal$(p_1,p_2) =$
$p_2\downarrow 1 \le p_2\downarrow 2 \rightarrow (p_2\downarrow 1 \le p_1\downarrow 1 \le p_1\downarrow 2 \le p_2\downarrow 2),$
$(p_2\downarrow 1 \ge p_1\downarrow 1 \ge p_1\downarrow 2 \ge p_2\downarrow 2)$

legal is a function of two nonnegative integer pairs p_1 and p_2. The pair p_2 is interpreted as a

declared indexing scheme. That is, $p_2\downarrow 1$ gives the leftmost index and $p_2\downarrow 2$ gives the rightmost index. The pair p_1 is interpreted as the reference. What the function does is to first determine whether the indexing scheme described by p_2 is ascending or descending and then to determine whether the pair p_1 is an appropriate reference given that scheme.

After determining that a reference is legal, the reference must be made. Because AMDL supports such a wide range of indexing schemes, references are first converted to *normalized* form. Normalized form is that which allows us to extract elements from a sequence using the standard operation "$\downarrow$". That is, we will assume sequences are indexed from left to right, beginning with 1, and that normalized references assume this indexing method. We introduce a normalization function ***norm*** which performs this normalization process.

(SF13) $\textbf{\textit{norm}}:(P\times P)\rightarrow P$ Normalizes access

$\textbf{\textit{norm}}(p_1,p_2) =$

$p_2\downarrow 1\leq p_2\downarrow 2\rightarrow\langle p_1\downarrow 1 - p_2\downarrow 1 + 1, p_1\downarrow 2 - p_2\downarrow 1 + 1\rangle,$

$\langle p_2\downarrow 1 - p_1\downarrow 1 + 1, p_2\downarrow 1 - p_1\downarrow 2 + 1\rangle$

for $\textbf{\textit{legal}}(p_1,p_2)$.

Given a reference p_1 and a defining pair p_2, ***norm*** returns a normalized reference. Rather than stepping through the definition, we will give a few examples

$\textbf{\textit{norm}}(\langle 7,0\rangle,\langle 7,0\rangle) = \langle 1,8\rangle$

$\textbf{\textit{norm}}(\langle 2,3\rangle,\langle 0,15\rangle) = \langle 3,4\rangle$

$\textbf{\textit{norm}}(\langle 2,3\rangle,\langle 1,4\rangle) = \langle 2,3\rangle$

$\textbf{\textit{norm}}(\langle 5,5\rangle,\langle 7,5\rangle) = \langle 3,3\rangle$

The actual extraction of elements from a sequence is done via the function ***extract***, defined as follows:

(SF8) $\textbf{\textit{extract}}:(P\times D^+)\rightarrow D^+$ Returns sub-sequence of D^+

$\textbf{\textit{extract}}(p,\delta) =$

$\langle\delta\downarrow(p\downarrow 1),\delta\downarrow((p\downarrow 1)+1),\ \ldots\ ,\delta\downarrow(p\downarrow 2)\rangle$

for $1\leq p\downarrow 1\leq p\downarrow 2\leq \#\delta$

Let us now examine the second type of variable reference.

(R3) $R[\textbf{(bits I K)}]\rho\gamma\eta =$

$\delta\notin \text{WORD}\rightarrow \textbf{\textit{err}}(11).$

$K[\text{K}]\rho\gamma\{\lambda p.\neg\textbf{\textit{legal}}(p,\delta\downarrow 2)\rightarrow\textbf{\textit{err}}(40),$

$\eta(\textbf{\textit{extract}}(\textbf{\textit{norm}}(p,\delta\downarrow 2),\delta\downarrow 1))\}$

where $\delta = \rho\downarrow 1[\text{I}]$

First, δ is checked to see that is has a WORD definition, as before. If this is the case, then $K[\text{K}]$ is evaluated with a structure reference continuation that contains the appropriate bounds checking. In this continuation, first the argument p is checked to see that it is a legal access.

This is done in ***legal***(p,δ↓2), which compares the pair p with the bounds from the environment δ↓2. If the reference is legal, then no more errors are possible and the original variable reference continuation may be applied to the appropriate subset of the location string for that variable. This subset is found by first normalizing the reference p (***norm***(p,δ↓2)) and then extracting the appropriate locations (***extract***(***norm***(p,δ↓2),δ↓1)).

To determine whether bit references outside of the defined boundaries are valid in this case, the function ***legal*** can be consulted. It will show that this is not allowed.

The next type of variable reference involves array accesses. Here, we must perform bounds checking as well, but rather than having a structure reference K involved, we have an expression E.

(R2) R**[(words** I E)]$\rho\gamma\eta$ =
  δ∉ARRAY→***err***(11),
  E**[E]**$\rho\gamma${λe.¬***legal***(⟨e↓1,e↓1⟩,δ↓3)→***err***(12),
    η(***extract***(***norm***(⟨e↓1,e↓1⟩,δ↓3),δ↓1)↓1)}
where δ = ρ↓1**[I]**

First, δ is checked, as it must be an ARRAY. Then, the expression E is evaluated with a continuation similar to the one for bit reference. Because ***legal*** expects number pairs, we form a pair ⟨e↓1,e↓1⟩ where e↓1 is the value component of the evaluated expression. We then call ***legal*** with this pair as one argument and the array bounds (δ↓3) as the other argument. If the access is legal, then we use ***extract*** and ***norm*** as we did with the bit reference, except in this case we are passing an array location string to ***extract*** rather than a word location string. The result is that we get back a list of one element, which is the word location string we want. This one element is then selected.

The final type of variable reference combines both a bit reference and a word reference. The equation for this type simply combines the operations of the previous equations.

(R1) R**[(words (bits** I K) E)]$\rho\gamma\eta$ =
  δ∉ARRAY→***err***(11),
  E**[E]**$\rho\gamma${λe.¬***legal***(⟨e↓1,e↓1⟩,δ↓3)→***err***(12),
    K**[K]**$\rho\gamma\mu$}
where δ = ρ↓1**[I]**
  μ = {λp.¬***legal***(p,δ↓2)→***err***(13),
    η(***extract***(***norm***(p,δ↓2),ω))}
where ω = ***extract***(***norm***(⟨e↓1,e↓1⟩,δ↓3),δ↓1)↓1

Here we see how the order of evaluation of the components of the construct, in this case the K and the E, is specified. By giving E**[E]** a continuation with includes K**[K]** we are specifying that E is to be evaluated first.

The question of specifying the order of evaluation of the components of a construct is a significant one. One clearly, then, has to make some statement concerning this evaluation. One possible statement would be to say that such order is undefined. This would heavily discourage the use of side-effect-producing constructs in certain situations, although it would admit the possibility of programs which give nonreproducible results. In addition, it would relieve the programmer from having to memorize possibly complex evaluation rules. On the other hand, we could explicitly state the order in which constructs are evaluated. This would permit "side-effect programming", provided one was willing to memorize the evaluation rules. It would also ensure programs which give the same result each time they are run. There are arguments for both approaches. Some language designers have avoided the problem by eliminating side-effects from important constructs, such as expressions. In this definition, we have chosen to specify the order of evaluation explicitly, partly due to the simplicity of this approach from a notational standpoint. However, there are techniques in denotational semantics which allow one to specify that the order of evaluation is random. These techniques are discussed in [Milne and Strachey].

4.3.6 Expressions (E:Exp→U→G→K→C)

By now, we have come to expect a certain kind of semantic domain. There is usually an environment, a procedure continuation, a continuation for the particular construct and a store. The semantic function E follows this convention, in this case using the expression continuation K. K, again, is the set of functions from expression values and stores into answers (E→C).

Before we can examine the equations however, we need to introduce some additional support functions which will allow us to move easily between expression values and bitstrings.

(SF5) ***econv***:B→E — Converts a B to an E

$$\textbf{econv}(\beta) = \langle n, \#\beta \rangle$$
$$\text{where } n = (\#\beta = 1 \rightarrow \beta\downarrow 1,\ \beta\downarrow(\#\beta) + 2*(e\downarrow 1))$$
$$e = \textbf{eco}\cdots(\textbf{extract}(\langle 1, \#\beta - 1\rangle, \beta))$$

(SF4) ***bconv***:E→B — Converts an E to a B

$$\textbf{bconv}(e) = \beta$$
$$\text{such that } \textbf{econv}(\beta) = e$$

(SF14) ***pwr***:(N×V)→B — Generates tuples of an element from V

$$\textbf{pwr}(n,v) = \langle v_1, v_2, \ldots, v_n \rangle$$
$$\text{where } v_i = v,\ i = 1,2,\ \ldots\ ,n$$
$$\text{for } n \geq 1$$

(SF3) ***adjust***:(B×N×V)→B — Shortens a bitstring, or extends it on the left with copies of a given element from V

$$\textbf{adjust}(\beta,n,v) =$$
$$n \leq \#\beta \rightarrow \textbf{extract}(\langle \#\beta - n + 1, \#\beta \rangle, \beta),$$

$$\mathbf{pwr}(n - \#\beta, v)\S\beta$$

The first two functions **econv** and **bconv** convert between expression values and bitstrings. The definition of **bconv** can simply be the inverse of **econv** because the functions are one-to-one. The function **pwr** generates all-zero or all-one bitstrings and is used by **adjust**, which either extends or truncates a bitstring on the left.

The first type of expression is used to extract a bitfield from or to extend another expression. The equation is

$$\begin{aligned}(E1)\ \ & E[\![(\mathbf{ussub}\ \ E\ \ K)]\!]\rho\gamma\kappa = \\ & \quad E[\![E]\!]\rho\gamma\{\lambda e.K[\![K]\!]\rho\gamma\mu\} \\ & \text{where}\ \ \mu = \{\lambda p.p\downarrow 1 \geq p\downarrow 2 \rightarrow \kappa(\mathbf{econv}(\mathbf{extract}(\mathbf{norm}(p,\langle n-1,0\rangle),\beta))), \\ & \qquad\qquad \mathbf{err}(14)\}, \\ & \qquad n = \mathbf{max}(p\downarrow 1 + 1, e\downarrow 2) \\ & \qquad \beta = \mathbf{adjust}(\mathbf{bconv}(e),n,0)\end{aligned}$$

E is evaluated with a continuation which includes the evaluation of K. $K[\![K]\!]$ is evaluated with a continuation that first checks that the resulting structure reference is legal. In this situation, "legal" means that the first number is greater than or equal to the second number, since there is an implicit indexing of the bits of the binary representation of expression values in this situation, which is ascending right-to-left beginning with zero. If the pair is legal, then the original expression continuation κ is applied to the appropriate value. This value is formed by first letting n be the maximum of the length of the expression and the leftmost number in p plus 1. We then create a bitstring of length n, with the assurance that we can then legally extract a subfield with the pair p. This is done by converting the expression to a bitstring with **bconv**(e), then passing this and the number n to the function **adjust** which creates a new bitstring β of length n, padded, if necessary, with the third argument 0. Bits are then extracted from β using the pair p normalized to the aforementioned indexing scheme (**extract**(**norm**(p,⟨n-1,0⟩),β)). This resulting bitstring is then converted to an expression value with **econv** and the result passed to the expression continuation κ.

The second type of expression involves the binary operators.

$$\begin{aligned}(E2)\ \ & E[\![(B\ \ E_1\ \ E_2)]\!]\rho\gamma\kappa = \\ & \quad E[\![E_1]\!]\rho\gamma\{\lambda e_1.E[\![E_2]\!]\rho\gamma\{\lambda e_1\sigma.B[\![B]\!](e_1,e_2)\{\lambda e.\kappa(e)\sigma\}\}\}\end{aligned}$$

E_1 is evaluated, followed by E_2 and both results passed to $B[\![B]\!]$. Note that B does not require an environment or a procedure continuation. This is because it operates only on expression values. It does require a continuation, however, because it may result in an error (e.g., division by zero). The continuation for binary operators is a function from expressions into answers. This is different from an expression continuation which is a function from an expression *and a store* into an answer. Consequently, we must construct the continuation $\{\lambda e.\kappa(e)\sigma\}$ for use with $B[\![B]\!]$.

The equation for expressions involving unary operators looks similar, except that the unary operator function U does not require a continuation. Rather it returns an expression value which is then used as an argument to the expression continuation. The equation is then

(E3) $E[\![(\mathrm{T}\ \ \mathrm{E})]\!]\rho\gamma\kappa =$
$\quad E[\![\mathrm{E}]\!]\rho\gamma\{\lambda e.\kappa(U[\![\mathrm{T}]\!](e))\}$

Before we look at the fourth type of expression, it is time that we examined the procedure continuation domain G a little more closely. Elements from G are used to represent something similar to a run-time stack of procedure invocations. It is actually a function from procedure pointers (locations) to a triple of "jumps". These jumps correspond to the *rest of the program* for a **leave**, **restart**, and **resume** of that procedure. This definition always uses the variable γ to represent that function. At any point in the computation, $\gamma(l)$ (where l is a location associated with a procedure) will be defined for exactly those procedures which are currently invoked. This latter fact is used in the following equation.

The fourth type of expression is a procedure call. Its definition requires the introduction of two more support functions.

(SF22) ***update***$:(D^+ \times N \times D) \rightarrow D^+$ Sequence update function
update$(\delta,n,d) =$
$\quad \langle \delta\downarrow 1, \delta\downarrow 2, \ \ldots\ , \delta\downarrow(n-1), d, \delta\downarrow(n+1), \ \ldots\ \delta\downarrow(\#\delta)\rangle$
for $1 \leq n \leq \#\delta$

(SF4.1) ***deref***$:(W \times S) \rightarrow E$ Dereference function
deref$(\omega,\sigma) =$
$\quad$ ***econv***$(\langle \sigma(\omega\downarrow 1), \sigma(\omega\downarrow 2), \ \ldots,\ \sigma(\omega\downarrow(\#\omega))\rangle)$
$\quad$ for $\#\omega \geq 1,\ \sigma(\omega\downarrow i) \in V,\ i = 1,2,\ldots,\omega$

The ***update*** function is used to update a sequence. The first argument δ is the sequence to be updated. The second argument n gives the index of the element in the sequence to be replaced, and the final argument d is the new element to be inserted. The function returns the updated sequence. For example, ***update***$(\langle a,b,c\rangle,2,d) = \langle a,d,c\rangle$. The dereference function simply interprets a word location string with respect to a given store.

With these support functions defined, we can now introduce the semantics of the procedure call.

(E4) $E[\![(\mathbf{call}\ \mathrm{P}\ \mathrm{E}_1\ \mathrm{E}_2\ \ldots\ \mathrm{E}_{n\geq 0})]\!]\rho\gamma\kappa\sigma_1 =$
$\quad \delta \in \varnothing \rightarrow$ ***err2***(9),
$\quad \delta\downarrow 2 \neq n \rightarrow$ ***err2***(10),
$\quad \gamma(\delta\downarrow 1) \notin \varnothing \rightarrow$ ***err2***(17),
$\quad E[\![\mathrm{E}_1]\!]\rho\gamma\{\lambda e_1 . E[\![\mathrm{E}_2]\!]\rho\gamma\{\lambda e_2 . \ \ldots\ E[\![\mathrm{E}_n]\!]\rho\gamma$
$\qquad \{\lambda e_n . \sigma_1(\delta\downarrow 1)(e_1,e_2,\ldots,e_n)\gamma_1\}\ \ldots\ \}\}\sigma_1$
where $\delta = \rho\downarrow 2(RI[\![\mathrm{P}]\!])$

$\gamma_1 = \gamma[\textbf{update}(\gamma(\rho\downarrow 4),3,j)/\rho\downarrow 4][\langle j,\oplus,\oplus\rangle/\delta\downarrow 1]$

$j = R[P]\rho\gamma\{\lambda\omega\sigma_2.\kappa(\textbf{deref}(\omega,\sigma_2))\sigma_2\}$

n is the subscript of E_n

if n = 0, then the last clause in the equation reduces to

$\sigma(\delta\downarrow 1)(\langle\rangle)\gamma_1$

The first thing that should be pointed out is that the above definition uses a new error function. The function ***err2*** is identical to ***err*** except that it has no store argument. As you may recall, we introduced the store argument into ***err*** simply as a convenience and that it was irrelevant to the definition of ***err***. We do not want a store argument in this case because, unlike previous situations, we are specifying a σ on the left-hand side of the equation. The σ appears on the left-hand side of the equation because it was necessary to reference it on the right-hand side.

Let's examine the above equation a line at a time. δ is defined to be $\rho\downarrow 2(R[\![P]\!])$, the procedure portion of the environment applied to the identifier contained in the variable reference. The first two lines of the equation check that the procedure is defined and that the number of formal and actual parameters match. The third line, $\gamma(\delta\downarrow 1)\notin\varnothing\rightarrow$***err2***(17), checks to see if the location associated with the procedure is defined in γ. If it were, then that would mean that a recursive procedure call is being attempted, which is not allowed.

If the function is defined and the call is not recursive, then each of the expressions is evaluated, beginning with the first. Then, the results $(e_1,e_2,\ldots,e_n)$ of the evaluation are passed to the procedure abstraction $\sigma_1(\delta\downarrow 1)$. (The location $(\delta\downarrow 1)$ provides an index into σ_1 which produces a procedure absraction of type F.) This result is then further applied to an updated procedure continuation, γ_1. ***update***$(\gamma(\rho\downarrow 4),3,j)$ yields an updated procedure continuation entry for the current procedure, whose location is given by $\rho\downarrow 4$. The change made is to replace the third (**resume**) component with the new jump j. This jump j represents what the rest of the program is to be for the case of a **resume** of this procedure. The procedure continuation is further updated by setting the *called* procedure's entry to the triple $\langle j,\oplus,\oplus\rangle$ which serves to define the **leave** component, leaving the other two entries undefined. (The **restart** component will get set when the procedure is actually entered.)

Finally, we need to look at the jump j. This jump will evaluate the variable reference with the *current* environment ρ, the *current* procedure abstraction γ, and a *new* store. (The use of the new store is implied due to the fact that the right hand side is a function of an unspecified store. If the old store were desired for the variable reference, then j would have had to be defined by $j = \{\lambda\sigma_3.R[\![P]\!]\rho\gamma\{\lambda\omega\sigma_2.\kappa(\textbf{deref}(\omega,\sigma_2))\sigma_2\}\sigma_1\}$. But this doesn't make much sense, because the new store is now lost!) The continuation passed to the variable reference takes a location string ω and a store σ_2 and dereferences the location string in that store (***deref***(ω,σ_2)), resulting in an expression value. This expression value is then passed to the original expression continuation κ and that result applied to the store σ_2.

The fifth type of expression is the assignment or transfer expression. Its semantic definition is

$$(E5)\quad E[((P_1\ P_1\ \ldots\ P_n)\ T\ E))]\rho\gamma\kappa =$$
$$E[E]\rho\gamma\{\lambda e.R[P_1]\rho\gamma\{\lambda\omega_1.R[P_2]\rho\gamma\{\lambda\omega_2.\ \ldots\ R[P_n]\rho\gamma\eta\ \ldots\ \}\}\}$$
$$\text{where}\quad \eta = \{\lambda\omega_n\sigma.\kappa e(T[T]((\omega_1\S\omega_2\S\ \ldots\ \S\omega_n),e)\sigma)\}$$

The variable references $P_1\ P_1\ \ldots\ P_n$ constitute the left-hand side of the transfer and the expression **E** constitutes the right. As there is more than one type of transfer in AMDL, the transfer operator has its own syntactic domain T. AMDL transfers allow multiple variable references in the destination, and there is no restriction that these variable references be disjoint. There are several ways in which this transfer could be defined, but let us first look at the above definition.

First, the right-hand side expression is evaluated with a continuation that includes the evaluation of P_1. The continuation of $R[P_1]$ includes the evaluation of P_2 and so on. The continuation η of the last variable reference applies the original expression continuation κ to the result of evaluating the right-hand expression. This result is further applied to the new store produced by $T[T]((\omega_1\S\omega_2\S\ \ldots\ \S\omega_n),e)\sigma$. The arguments to $T[T]$ are the concatenation $\omega_1\S\omega_2\S\ \ldots\ \S\omega_n$ of the location strings produced by the variable references, the result **e** of evaluating the right-hand side expression, and the current store σ.

While we would need to examine the definition of T in order to determine how the bits of the expression are assigned to the left-hand side, there are many observations that can be made from the above equation. First of all, the order of evaluation of the components of the construct is **E**, P_1, P_2, ... ,P_n. Secondly, it can be seen that no assignment takes place until all of the left-hand side has been evaluated. This is important because an individual assignment could affect the evaluation of the remaining variable references. Finally, we see that the original expression continuation is applied to the expression value returned by the right-hand side. Another possible interpretation would be to dereference the left-hand side after making the assignment and to use that dereferenced value as the argument to the expression continuation. This dereferenced value could, of course, be different from the value of the right-hand side.

The next expression type we look at is the variable reference. The meaning of a variable reference as an expression is simply the dereferenced value of the variable. The equation is

$$(E6)\quad E[P]\rho\gamma\kappa = R[P]\rho\gamma\{\lambda\omega\sigma.\kappa(\mathbf{deref}(\omega,\sigma))\sigma\}$$

Note that dereferencing must be done with respect to a particular store.

Finally, an expression can be a constant. The semantics are defined by

$$(E7)\quad E[X]\rho\gamma\kappa = \kappa(C[X])$$

The term $C[X]$ yields an expression value which is used as an argument to the expression continuation κ.

4.3.7 Transfer Operators (T:Tra→(W×E)→S→S)

We just examined the role of the transfer operator in a transfer expression. Its semantics is to produce a new store from an old one by updating a string of locations with a particular expression value. Before we examine the equations for transfer operators, though, we need two new support functions which will allow us to express the updating of stores more concisely.

(SF17) ***setb***:(W×B)→S→S Updates an S with a B
$\mathbf{setb}(\omega,\beta)\sigma = \sigma[\beta\downarrow 1/\omega\downarrow 1][\beta\downarrow 2/\omega\downarrow 2]\ \ldots\ [\beta\downarrow(\#\beta)/\omega\downarrow(\#\omega)]$
for $\#\omega = \#\beta$ and $\#\omega \geq 1$

(SF18) ***setba***:(W×B)→S→S Updates an S with an *adjusted* B
$\mathbf{setba}(\omega,\beta) = \mathbf{setb}(\omega,\mathbf{adjust}(\beta,\#\omega,0))$

The function ***setb*** is used to update a store, given a location string and a bitstring. Both strings must be of equal length. Note the semantics of the "[]" operator tells us that the locations are updated from left to right. This is important because the locations within a location string need not be unique. The function ***setba*** is identical to ***setb*** except that its bitstring argument need not be the same length as the location string argument. If the two lengths differ, the bitstring argument is adjusted before the update takes place.

There are two transfer operators, corresponding to the two arithmetic modes of AMDL, unsigned and two's-complement. The first operator is ***usset*** and is defined by

(T1) $T[\mathbf{usset}](\omega,e) = \mathbf{setba}(\omega,\mathbf{bconv}(e))$

The semantics of this operator is just the semantics of ***setba*** except that the value argument is an expression.

The second transfer operator is ***tcset*** and is defined by

(T2) $T[\mathbf{tcset}](\omega,e) = \mathbf{setb}(\omega,\mathbf{adjust}(\beta,\#\omega,1))$
where $\beta = \mathbf{bconv}(e)$

In this case, the value is first converted to a bitstring, then adjusted using its leftmost bit as a extension value, rather than zero as in the case of ***usset***. By substituting the definition of ***setba*** in the definition of ***usset***, one can see precisely the difference in semantics of the two transfer operators.

Through the use of the ***setb*** and ***setba***, the semantics of the transfer operators specifies that the order of assignment of values to locations is left-to-right. This need not be the case, of course. A right-to-left ordering or an unspecified ordering are two other equally valid possibilities.

4.3.8 Structures (S:Str→P)

A structure is used in several places in AMDL. Primarily, it is used to define the bit or word structure of a variable. It also is found in a particular type of action, the **decode** action, which

we will look at shortly. Its semantics is simply a pair of numbers, corresponding to the constant(s) present

(K1) $S[\textbf{(pair } X_1 \ X_2)] = \langle C[X_1]\downarrow 1, C[X_2]\downarrow 1\rangle$

(K2) $S[X] = \langle C[X]\downarrow 1, C[X]\downarrow 1\rangle$

In the case of (K2) where a single constant is specified, the value component is duplicated. Note that the length component of the constants is ignored in both cases.

4.3.9 Actions $(A{:}Act\rightarrow U\rightarrow G\rightarrow C\rightarrow C)$

Actions provide the control structures for AMDL. Like the expressions, their semantics is a function of an environment ρ, a procedure continuation γ, and their own continuation. An action continuation is simply a function $S\rightarrow ANS$ from stores into answers. This says that the intermediate result of an action is just a new store, and that actions do not produce any other values (as do expressions).

The first action we will look at is the **label** action. Before we examine the definition, let us first consider what the role of a label is in AMDL. Labels can be referred to by two operators, **leave** and **restart,** which are used to alter the otherwise sequential course of execution. Briefly, a **leave** of a labelled action is like a "goto" to the end of the action and a **restart** is like a "goto" to the beginning of that action. In order that these **leave** and **restart** operators can function, a pair of "jump" continuations, one for each operator, is placed in the ULAB (third) component of the environment, which is a function of type $Ide\rightarrow((J\times J)+\oplus)$, where J is the domain of jump continuations with $J = C$. Given an environment ρ, therefore, and label identifier I, then $\rho\downarrow 3\downarrow 1$ would give the semantics of a **leave** of that label and $\rho\downarrow 3\downarrow 2$ the semantics of a **restart**. We are now ready to examine the **label** definition.

(A3) $A[\textbf{(label I A)}]\rho\gamma\theta =$

$$A[A]\textit{update}(\rho,3,\rho\downarrow 3[\langle\theta, A[\textbf{(label I A)}]\rho\gamma\theta\rangle/I])\gamma\theta$$

The semantics of the **label** action is the semantics of the enclosed action A with an updated environment $\textit{update}(\rho,3,\rho\downarrow 3[\langle\theta, A[\textbf{(label I A)}]\rho\gamma\theta\rangle/I])$. The third element of the environment ρ, as stated earlier, is used in conjunction with **leave** and **restart** actions. Here, the I component of the third element is being replaced with the sequence $\langle\theta, A[\textbf{(label I A)}]\rho\gamma\theta\rangle$. The first element θ of this sequence is the "leave" component of the list. That is to say, the semantics of a **leave** action within the scope of A will be the semantics θ of the rest of the program following this **label** statement. The second component $A[\textbf{(label I A)}]\rho\gamma\theta$ of the list is the "restart" component. This says that the semantics of a **restart** action within the scope of the action A is the semantics of the **label** action itself. This recursive definition is quite natural when one views the **restart** operator as a "goto" back to the start of the **label** action. Note that the both terms in the sequence $\langle\theta, A[\textbf{(label I A)}]\rho\gamma\theta\rangle$ are themselves action continuations, that is, functions from a store into an answer. This is necessary, of course, so that the store at the time of the **restart** or **leave** is passed along to the remaining computation.

There is no **resume** element in the sequence because a **resume** of a label is never legal.

The semantics of the **leave**, **restart** and **resume** operators are now relatively straightforward. Recall that procedure continuations γ contain the continuations associated with a **leave**, **restart** or **resume** of *procedures* which are currently active. There must be some algorithm, therefore, whereby these operators determine whether they refer to labels or procedures. Because **resume** is not defined to operate with respect to labels, it will always refer to procedures. The operators **leave** and **restart**, however, will refer to labels if the label encloses the operator in the current procedure. Only if no such label exists, will the **leave** and **restart** operators refer to procedures. The semantics, then, are as follows:

(A4) $A[\![(\textbf{leave}\ \ \mathrm{I})]\!]\rho\gamma\theta =$
$\quad \rho\downarrow 3[\![\mathrm{I}]\!] \notin \varnothing \rightarrow \rho\downarrow 3[\![\mathrm{I}]\!]\downarrow 1,$
$\quad \delta \in \varnothing \rightarrow \mathit{err}(7),$
$\quad \gamma(\delta\downarrow 1) \in \varnothing \rightarrow \mathit{err}(8),$
$\quad \gamma(\delta\downarrow 1)\downarrow 1$
$\quad$ where $\delta = \rho\downarrow 2[\![\mathrm{I}]\!]$

(A5) $A[\![(\textbf{restart}\ \ \mathrm{I})]\!]\rho\gamma\theta =$
$\quad \rho\downarrow 3[\![\mathrm{I}]\!] \notin \varnothing \rightarrow \rho\downarrow 3[\![\mathrm{I}]\!]\downarrow 2,$
$\quad \delta \in \varnothing \rightarrow \mathit{err}(7),$
$\quad \gamma(\delta\downarrow 1) \in \varnothing \rightarrow \mathit{err}(8),$
$\quad \gamma(\delta\downarrow 1)\downarrow 2$
$\quad$ where $\delta = \rho\downarrow 2[\![\mathrm{I}]\!]$

(A6) $A[\![(\textbf{resume}\ \ \mathrm{I})]\!]\rho\gamma\theta =$
$\quad \delta \in \varnothing \rightarrow \mathit{err}(7),$
$\quad \gamma(\delta\downarrow 1) \in \varnothing \rightarrow \mathit{err}(8),$
$\quad \gamma(\delta\downarrow 1)\downarrow 3 \in \varnothing \rightarrow \mathit{err}(8),$
$\quad \gamma(\delta\downarrow 1)\downarrow 3$
$\quad$ where $\delta = \rho\downarrow 2[\![\mathrm{I}]\!]$

Both the **leave** and **restart** definitions first check whether or not there is a valid label of that name by checking the value of $\rho\downarrow 3[\![\mathrm{I}]\!]$. If this value is not undeclared, then it is a list of two continuations and the appropriate continuation is extracted to provide the semantics of the action. If no such label exists, then the action must refer to a procedure. All three definitions define δ to be the procedure element of the environment applied to the identifier I. If $\delta \in \varnothing$, then no procedure has been associated with that identifier, and an error has occurred. If $\delta \notin \varnothing$, however, then $\delta\downarrow 1$ is a location which can be used as an index into γ. If $\gamma(\delta\downarrow 1) \in \varnothing$, then the procedure is not active and an error has occurred. If the procedure is active, however, then $\gamma(\delta\downarrow 1)$ will be a triple of "jumps" corresponding to **leave**, **restart** and **resume** of that procedure. Actually, the third jump (for **resume**) will be undeclared if the procedure referenced is the current procedure, and an error will occur. In all other cases, however, the jump is simply extracted as the value of the action.

It should be noted that in determining the procedure location $\delta\downarrow 1$, the environment is consulted, while the actual jump is obtained from the procedure continuation γ. This means that only procedures which are statically accessible may be referenced by these operators. Procedures outside the static scope of the operator can never be left, restarted or resumed.

The **call** action is very close to the **call** expression, both in syntax and semantics. The only difference in syntax is that the **call** expression contains a variable reference and the **call** action only contains an identifier. The semantics differ in that the **call** expression effects a reference to the variable after the procedure has been invoked, whereas the **call** action does not. The equation is

$$
\begin{aligned}
\text{(A10)}\ & E[\textbf{(call I } E_1\ E_2\ \ldots\ E_{n\geq 0})]\rho\gamma\theta\sigma_1 = \\
& \quad \delta\in\varnothing\rightarrow \textbf{err2}(9), \\
& \quad \delta\downarrow 2\neq n\rightarrow \textbf{err2}(10), \\
& \quad \gamma(\delta\downarrow 1)\notin\varnothing\rightarrow \textbf{err2}(17), \\
& \quad E[E_1]\rho\gamma\{\lambda e_1 . E[E_2]\rho\gamma\{\lambda e_2 \cdot\ \ldots\ E[E_n]\rho\gamma \\
& \qquad \{\lambda e_n . \sigma_1(\delta\downarrow 1)(e_1,e_2,\ldots,e_n)\gamma_1\}\ \ldots\ \}\}\sigma_1 \\
& \text{where } \delta = \rho\downarrow 2[1] \\
& \quad \gamma_1 = \gamma[\textbf{update}(\gamma(\rho\downarrow 4),3,j)/\rho\downarrow 4][\langle j,\oplus,\oplus\rangle/\delta\downarrow 1] \\
& \quad j = \theta \\
& \quad n \text{ is the subscript of } E_n \\
& \quad \text{if } n=0, \text{ then the last clause in the equation reduces to} \\
& \qquad \sigma(\delta\downarrow 1)(\langle\rangle)\gamma_1
\end{aligned}
$$

Note that the main difference in this equation and (E4) for the **call** expression is in the definition of j. In this case, j is simply the continuation of the action itself.

The semantics of the **repeat** and **seq** is straightforward. The equations are

$$
\begin{aligned}
\text{(A7)}\ & A[\textbf{(repeat A)}]\rho\gamma\theta = \\
& \quad A[A]\rho\gamma\{A[\textbf{(repeat A)}]\rho\gamma\theta\} \\
\text{(A8)}\ & A[\textbf{(seq } A_1\ A_2\ \ldots\ A_n)]\rho\gamma\theta = \\
& \quad A[A_1]\rho\gamma\{A[A_2]\rho\gamma\ \ldots\ \{A[A_n]\rho\gamma\theta\}\ \ldots\ \}
\end{aligned}
$$

The first equation says simply that the semantics of **repeat** is the semantics of the enclosed action with a continuation equal to the semantics of the **repeat** itself. The situation here is identical to constructing the **restart** component in the **label** action. This is reasonable, as a **repeat** is equivalent to a **label** with a **restart** as the last embedded action. The semantics of **seq** shows a nested set of continuations producing the sequential execution of the individual actions.

The **par** action is supposed to indicate parallel execution of a set of actions. We have not worked out the details of this semantics, however, so for the moment the semantics is identical to

the semantics of the **seq** action.

(A9) $A[\![(\textbf{par}\ A_1\ A_2\ \ldots\ A_n)]\!]\rho\gamma\theta =$
$A[\![A_1]\!]\rho\gamma\{A[\![A_2]\!]\rho\gamma\ \ldots\ \{A[\![A_n]\!]\rho\gamma\theta\}\ \ldots\ \}$

The **cond** action is like a limited McCarthy conditional. Its semantics is

(A1) $A[\![(\textbf{cond}\ (E\ A))]\!]\rho\gamma\theta = E[\![E]\!]\rho\gamma\{\lambda e.e\downarrow 1 \neq 0 \rightarrow A[\![A]\!]\rho\gamma\theta, \theta\}$

The expression **E** is evaluated with a continuation which either evaluates the action A (if the value part of the result of the expression is non-zero) or is the normal continuation θ.

Since expressions are also legal actions, the semantics of an "expression" action is given by

(A11) $A[\![E]\!]\rho\gamma\theta = E[\![E]\!]\rho\gamma\{\lambda e.\theta\}$

The fact that the expression value produced by $E[\![E]\!]$ is ignored is obvious from the continuation $\{\lambda e.\theta\}$.

The next action we consider is the **write** action. This action evaluates an expression and concatenates the value of the expression with the answer produced by the rest of the program. Its definition is one of the two (the other being the definition for **err**) where the elements from the answer domain **ANS** are actually manipulated. The equation is

(A12) $A[\![(\textbf{write}\ E)]\!]\rho\gamma\theta = E[\![E]\!]\rho\gamma\{\lambda e\sigma.\langle e\downarrow 1\ \text{in}\ \text{OUT}\rangle \S\theta(\sigma)\}$

The expression **E** is evaluated with a continuation $\{\lambda e\sigma.\langle e\downarrow 1\rangle \S\theta(\sigma)\}$ which specifies how the result is combined with the answer produced by the rest of the program. Normally, expression continuations are written as $\{\lambda e.\ \ldots\ \}$ where the body of the lambda expression is some function in $S \rightarrow ANS$. In the above continuation, however, we need to specify the store σ on the left-hand side of the lambda expression, because it must be referenced explicitly in the body. In the body, the action continuation θ is applied to the current store σ to produce an answer. This answer is then appended to the value portion $e\downarrow 1$ of the the expression evaluation to produce the new answer $\langle e\downarrow 1\ \text{in}\ \text{OUT}\rangle \S\theta(\sigma)$.

The **decode** action is the final action in AMDL. This action has been left until last because it contains a rather complicated component part. called a decode clause. A **decode** statement is similar to a *select* or *case* statement in other languages. The syntax consists of an expression and a list of decode clauses. The expression is first evaluated, and then the decode clauses are analyzed until one of them "succeeds." If none of the decode clauses succeeds, then the action is in error. The equation looks like

(A2) $A[\![(\textbf{decode}\ E\ (Z_1\ Z_2\ \ldots\ Z_n))]\!]\rho\gamma\theta =$
$E[\![E]\!]\rho\gamma\{\lambda e.Z[\![Z_1]\!](e\downarrow 1)(0)\{Z[\![Z_2]\!](e\downarrow 1)(1)\ \ldots$
$\{Z[\![Z_n]\!](e\downarrow n)(n\text{-}1)\{err(16)\}\rho\gamma\theta\}\ \ldots\ \rho\gamma\theta\}\rho\gamma\theta\}$

It's rather hard to understand the semantics of the **decode** action without understanding the

semantics of the decode clause, but the reverse situation is difficult as well. What the above equation says is that the expression E is evaluated with a continuation that contains the evaluation of the first decode clause. The arguments to that first decode clause are the result e↓1 of evaluating the expression E, the integer index of the decode clause in the **decode** action, a continuation $\{Z[Z_2] \ldots\}$ which will be evaluated if the decode clause *fails*, and the standard $\rho\gamma\theta$ which provides the environment, procedure continuation and action continuation. The action continuation θ is used if the decode clause *succeeds*. Note that the failure continuation passed to the final decode clause is {*err*(16)} which causes the action to fail if evaluated. If it was ever decided to change the semantics of an unsuccessful **decode** to a no-op, this failure continuation could simply be replaced by θ.

4.3.10 Decode Clauses (Z:Dcl→N→N→C→U→G→C)

With the semantics of the **decode** action fresh in our memory, let's examine the semantics of the decode clause. There are three types of decode clauses which differ in how they specify their "triggering" values. The first type of decode clause is always satisfied. Its semantics is

(Z4) $Z[\textbf{(otherwise A)}]nm\theta = A[A]$

The semantics is independent of the expression result n, the position m, and the failure continuation θ. The action A is always evaluated and is passed the success continuation. The above notation is an extreme example of how currying reduces the size of an equation.

The second decode clause we will examine specifies the triggering values by means of a single structure Σ. Its semantics is

(Z2) $$Z[(\Sigma\ A)]nm\theta_1\rho\gamma\theta_2 = S[\Sigma]\downarrow 1 \le n \le S[\Sigma]\downarrow 2 \rightarrow A[A]\rho\gamma\theta_2, \theta_1$$

If the value of n lies in the closed interval defined by the number pair $S[\Sigma]$, then the decode clause has "succeeded" and the action A is evaluated. Otherwise, the failure continuation θ_1 is taken.

The third type of decode clause is similar to that in (Z2) above, except that rather than specifying a single structure, a list of structures is given. In this case, the decode clause succeeds if the expression value falls within the closed interval defined by any of the structures. The semantic equation is

(Z1) $$Z[((\Sigma_1\ \Sigma_2\ \ldots\ \Sigma_n)\ A)]nm\theta_1\rho\gamma\theta_2 = (S[\Sigma_1]\downarrow 1 \le n \le S[\Sigma_1]\downarrow 2) \vee (S[\Sigma_2]\downarrow 1 \le n \le S[\Sigma_2]\downarrow 2) \vee \ \ldots\ \vee (S[\Sigma_n]\downarrow 1 \le n \le S[\Sigma_n]\downarrow 2) \rightarrow A[A]\rho\gamma\theta_2, \theta_1$$

None of the previous three types of decode clauses has utilized the "index" argument m of the

decode clause function. The final type of decode clause uses this value as an implicit structure with which to compare the expression value. The semantics is

(Z3) $Z[A]nm\theta_1\rho\gamma\theta_2 =$
$n = m \rightarrow A[A]\rho\gamma\theta_2,$
θ_1

Note that the term $n = m$ was used instead of the equivalent comparison $m \leq n \leq m$.

4.3.11 Variables (P:Var→N→X→X, PO:Var→N→U→O→O)

Variables are used within declarations. They may stand alone, in which case they act as their own declaration. They may also occur within a procedure declaration or an overlay statement. The semantics for variables must be general enough to handle each of these cases.

Before we can proceed further, we need to introduce yet another continuation, the declaration continuation. This is a function χ:X = Q→U→C, where Q is a domain of local binding functions, U is the domain of environments, and C contains our familiar action continuations. Recall from Section 4.2.5 that $q \in Q$ is a function which keeps track of which identifiers have been declared within the current context. The meaning of a variable is to take a binding function q, an environment ρ, a store σ and a declaration continuation χ and apply that declaration continuation to updated q, ρ and σ. The semantic domain for Var could then be written as P:Var→(Q×U×S×X)→ANS. But by rearranging the parameters and currying, we can obtain P:Var→X→X. The extra parameter N seen above is used to distinguish between the various contexts in which the variable occurs. The other semantic domain PO deals specifically with overlays and will be discussed later.

We also need to introduce two support functions **new** and **get** which allows us to allocate new locations from a store. **new** is defined simply by

(SF12) ***new***:S→L — Gets new location from an S

$$\textbf{\textit{new}}(\sigma) = \text{l such that } \sigma(l) = \oplus$$

For a given store σ, therefore, ***new***(σ) yields an "unused" location in σ. The function ***get*** is simlar to ***new***, except that ***get*** is used to obtain a *sequence* of unused locations from a store. It is defined as

(SF9) ***get***:(N×S)→L — Gets a sequence of locations from an S

$$\textbf{\textit{get}}(n,\sigma) = n = 1 \rightarrow \langle \textbf{\textit{new}}(\sigma)\rangle, \langle \textbf{\textit{new}}(\sigma)\rangle \S \textbf{\textit{get}}(n-1, \sigma[0/\textbf{\textit{new}}(\sigma)]) \quad \text{for } n \geq 1$$

Let us look at the semantics of the simple variable.

(P3) $$P[(\textbf{bits I } \Sigma)]n\chi q\rho\sigma_1 = q[I] = \text{true} \rightarrow \textbf{\textit{err2}}(13), \chi q[\text{true}/I]\textbf{\textit{update}}(\textbf{\textit{update}}(\rho,1,\rho\downarrow 1[\langle \omega, S[\Sigma]\rangle/I]),2,\rho\downarrow 2[\oplus/I])\sigma_2$$

where $\omega = (n = 2 \rightarrow \oplus, \textbf{\textit{get}}(n_1,\sigma_1))$

$\sigma_2 = (n = 2 \rightarrow \sigma_1, \textbf{\textit{setba}}(\omega,\langle 0\rangle)\sigma_1)$

$n_1 = \textbf{\textit{size}}(S[\Sigma])$

First, the function q is applied to the identifier I. If I has been previously declared, then an error has occurred. If this is not the case, then the declaration continuation χ is applied to an updated q, ρ and σ. The updated q is simply the same q with q[I] defined to be true. The new ρ has updated entries in the first and second components, corresponding to a new meaning for "I" as a variable and a procedure. The procedure component is updated with the undeclared element, $\oplus$. (In the case where this variable is nested within a procedure declaration, this entry will later be replaced.) The variable component of the environment and the new store depend on the argument n defining the context in which the variable occurs.

There are three possible values for n, with meanings that are given below.

0	Variable occurs by itself
1	Variable occurs within **proc** declaration
2	Variable occurs within **over** declaration

In the definition we are currently examining, a distinction is made between $n=2$ and $n\neq 2$. In both cases the structure $S[\Sigma]$ of the variable is placed in the appropriate field of the variable entry. The location string ω, however, depends on the value of n. If n is not equal to 2, then unused locations are obtained from the store σ and used for ω. If n is 2, then the location string in the variable entry for "I" is set to the undeclared element. This is due to the fact that in **over** declarations, no new locations are implied: variables are simply mapped onto existing locations. The new store σ_2 is simply the old store σ_1, in the case of the overlay. When new locations have been obtained from the store, however, these locations must be mapped into something other than $\oplus$ to avoid their possible re-use. This is done with ***setba***$(\omega,\langle 0\rangle)\sigma_1$, in the case of $n\neq 2$.

The next type of variable we will look at is the array variable. Its semantics is quite similar, except for the complications introduced by the additional dimension. In order to deal with this added dimension, we need to introduce another two support functions, ***size*** and ***reshape***. They are defined by

(SF19) ***size***:P→N — Gives "size" of a P

size(p) = ***abs***(p↓1-p↓2) + 1

(SF16) ***reshape***:(N×W)→A — Reshapes a W into an A

reshape(n,ω) =

$\langle\langle\alpha\downarrow 1,\alpha\downarrow 2, \ldots ,\alpha\downarrow m\rangle,$

$\langle\omega\downarrow(m+1),\omega\downarrow(m+2), \ldots ,\omega\downarrow(2*m)\rangle,$

.....

$\langle\omega\downarrow((n-1)*m+1),\omega\downarrow((n-1)*m+2), \ldots ,\omega\downarrow(n*m)\rangle\rangle$

where $m = (\#\omega)/n$

for $n\geq 1$ and $(\#\omega \bmod n) = 0$

size is used to yield the number of elements implied by a particular indexing pair. For example, ***size***$(\langle 0,7\rangle) = 8$. ***reshape*** is similar to the dyadic APL reshape operator ρ. Given a string ω

and a nonnegative integer n, it yields a sequence of n elements, each element containing successive $\#\omega/n$ elements from ω.

Looking at the definition for array variable declarations, then, we have

(P1) $P[\![(\textbf{words}\ (\textbf{bits}\ I\ \Sigma_1)\ \Sigma_2)]\!]n\chi q\rho\sigma_1 =$

$q[I] = \text{true} \rightarrow \textbf{err2}(2),$

$\chi q[\text{true}/I]\textbf{update}(\textbf{update}(\rho,1,\rho\downarrow 1[\langle\alpha,S[\Sigma_1],S[\Sigma_2]\rangle/I]),2,\rho\downarrow 2[\oplus/I])\sigma_2$

where $\alpha = (n = 2 \rightarrow \oplus, \textbf{reshape}(\textbf{size}(S[\Sigma_1]),\omega))$

$\sigma_2 = (n = 2 \rightarrow \sigma_1, \textbf{setba}(\omega,\langle 0\rangle)\sigma_1)$

$\omega = \textbf{get}(m,\sigma_1)$

$m = \textbf{size}(S[\Sigma_1]) * \textbf{size}(S[\Sigma_2])$

The major difference in this equation is in the definition of the location string ω and in the fact that both the word and bit structure of the variable are placed in the environment. The string ω is first formed by obtaining the appropriate number of locations from the old store. This string is then shaped into an array location string, using the ***reshape*** function. As before, the locations obtained must be marked as used in the new store.

The next variable type consists solely of an identifier "I". Its semantics are

(P4) $P[\![\mathbf{I}]\!]n\chi q\rho =$

$q[I] = \text{true} \rightarrow \textbf{err}(2),$

$n \neq 1 \rightarrow \textbf{err}(3),$

$\chi q[\text{true}/I]\textbf{update}(\textbf{update}(\rho,1,\rho\downarrow 1[\oplus/I]),2,\rho\downarrow 2[\oplus/I])$

As in (P1) and (P3) above, the value of **q[I]** is checked. Unless the variable occurs within a procedure declaration (n = 1), an error has occurred. This is due to the fact that in the case of the variable declaration or the overlay, the purpose of the variable is to define a new storage entity. Without any structure components, the variable cannot be properly defined. Note that the language could have defined a default bit structure for this case, in which case the definition would have appeared similar to definition (P3) for simple bit structured variables. If the variable occurs within a procedure declaration, however, the identifier will have some purpose, that of denoting the procedure associated with it. In that case, the environment is simply updated with the undeclared element $\oplus$ in both the variable and procedure components.

The remaining variable type is always an error when it occurs as a declaration. This is the case where a word structure is given without an accompanying bit structure. The semantics is simply

(P2) $P[\![(\textbf{words}\ I\ \Sigma)]\!]n\chi q\rho = \textbf{err}(3)$

We now come to the semantic function *PO*. As was said earlier, this function is used solely in conjunction with the overlay construct. The syntax of an overlay is quite general. A new variable is somehow equated with a list of previously defined variables. There are, of course, many possible interpretations of this general mechanism. The interpretation we have chosen may

not coincide with the reader's intuitive notion. It serves to show, however, that even the most complicated of interpretations can be precisely specified by this definitional method.

The basic idea behind our interpretation of the overlay is that the location strings associated with the previously defined variables are *individually* reshaped to coincide with the length of the new variable. These reshaped strings are then row-wise concatenated to form a new structure which must match exactly the dimensions of the new variable.

An example is in order. Consider the following collection of AMDL declarations

```
(bits old1 (pair 3 0))
(bits (words old2 (pair 0 5)) (pair 1 0))
(bits (words old3 (pair 0 1)) (pair 1 0))
(over (words new (pair 0 3)) (pair 4 0) (a b c)
```

or the equivalent ISPS declarations

```
old1<3:0>,
old2[0:5]<1:0>,
old3[0:1]<1:0>,
new: = old1@old2@old3
```

Then the following figure shows that the locations associated with the variables old1, old2 and old3 are combined together to define the variable new.

```
                                                | 5  6|
|1  5  6  7 17|                                 | 7  8|
|2  8  9 10 18| <==>    |1  2  3  4|            | 9 10|   |17 18|
|3 11 12 13 19|                                 |11 12|   |19 20|
|4 14 15 16 20|                                 |13 14|
                                                |15 16|
   (new)          <==>         old1             old2       old3
```

In this case a 4 by 5 array is being defined as the concatenation of a 4 bit word, a 6 by 2 array and a 2 by 2 array. Note that the locations for each of the variables old1, old2 and old3 fit exactly into one or more columns of the left hand side. This is a requirement of all overlays. Note that if this rule is obeyed and the total number of locations in the defining expression is equal to the number of locations required for the new variable, then the resulting array obtained from concatenating the arrays in the defining expression will always have a word length equal to that of the new variable. This is, in fact, the algorithm used to check the legality of an overlay in the ensuing definitions.

The formal definition of overlays is split between the definition of *PO* and the definition of declarations. The function *PO*, with which we are now concerned, is applied only to variables in the defining expression of an overlay. Given the length of the new variable, the intermediate result of *PO* is an array location string reshaped to that given length. We introduce the *overlay*

continuation o:O = A→ANS which accepts this reshaped string as its argument. We must use a continuation, because an error may be discovered within *PO*. In particular, a variable in the defining expression might not reshape evenly into the given length.

As we have seen, there are four variable types to be examined. Rather than starting with the simplest case, we will first treat the most difficult one. Once this definition is understood, however, the other cases will follow easily from it. First, however, we must define another support function, ***aconc***. ***aconc*** yields the row-wise concatenation of two array location strings and is defined as follows.

(SF2) ***aconc***:(A×A)→A Row concatenation of two As

$$\textbf{aconc}(\alpha_1,\alpha_2) = \langle \alpha_1\downarrow 1 \S \alpha_2\downarrow 1, \alpha_1\downarrow 2 \S \alpha_2\downarrow 2, \ \ldots \ , \alpha_1\downarrow(\#\alpha_1) \S \alpha_2\downarrow(\#\alpha_2)\rangle$$

for $\#\alpha_1 = \#\alpha_2$

Examining (PO1), then we have

(PO1) $PO[(\textbf{words (bits I } \Sigma_1) \ \Sigma_2)]n\rho o\alpha =$

$\delta \notin$ ARRAY→***err2***(5),

$\neg(\textbf{legal}(S[\Sigma_1],\delta\downarrow 2) \wedge \textbf{legal}(S[\Sigma_2],\delta\downarrow 3)) \rightarrow \textbf{err2}(5)$,

$(\textbf{size}(S[\Sigma_1]) * \textbf{size}(S[\Sigma_2]) \bmod n) \neq 0 \rightarrow \textbf{err2}(6)$,

$o(\textbf{aconc}(\alpha, \textbf{reshape}(n, \omega_{p\downarrow 1} \S \omega_{p\downarrow 1+1} \S \ \ldots \ \S \omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \textbf{extract}(\textbf{norm}(S[\Sigma_1],\delta\uparrow 2),\delta\downarrow 1\downarrow i)$

$p = \textbf{norm}(S[\Sigma_2],\delta\downarrow 3)$

The first thing that must be true is that the identifier referenced corresponds to an array variable. Second, the structures Σ_1 and Σ_2 must be legal references. Third, the number of bits in the referenced portion of the variable, computed by $\textbf{size}(S[\Sigma_1]) * \textbf{size}(S[\Sigma_2])$, must be an even multiple of the new variable length n. If all this is true, then the overlay continuation o is applied to the concatenation of the existing array location string α and the reshaped structure. This reshaping is done by first concatenating together the appropriate bitstrings in the appropriate words. The appropriate bitstrings are found by $\omega_i = \textbf{extract}(\textbf{norm}(S[\Sigma_1],\delta\downarrow 2),\delta\downarrow 1\downarrow i)$. The range of words referenced is given by $p = \textbf{norm}(S[\Sigma_2],\delta\downarrow 3)$.

The next variable type specifies a word range but does not give a bit range. Its semantics is:

(PO2) $PO[(\textbf{words I } \Sigma)]n\rho o\alpha =$

$\delta \notin$ ARRAY→***err2***(5),

$\neg \textbf{legal}(S[\Sigma],\delta\downarrow 3) \rightarrow \textbf{err2}(5)$,

$(\textbf{size}(\delta\downarrow 2) * \textbf{size}(S[\Sigma]) \bmod n) \neq 0 \rightarrow \textbf{err2}(6)$,

$o(\textbf{aconc}(\alpha, \textbf{reshape}(n, \omega_{p\downarrow 1} \S \omega_{p\downarrow 1+1} \S \ \ldots \ \S \omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \delta\downarrow 1\downarrow i$

$p = \textbf{norm}(S[\Sigma],\delta\downarrow 3)$

This definition is identical to the previous definition, with the exception that instead of using a user-provided structure Σ for the bit reference, the full bit range is assumed. The equation $\omega_i = \mathbf{extract}(\mathbf{norm}(S[\Sigma_1],\delta\downarrow 2),\delta\downarrow 1\downarrow i)$ then becomes $\omega_i = \mathbf{extract}(\mathbf{norm}(\delta\downarrow 2,\delta\downarrow 2),\delta\downarrow 1\downarrow i)$, which further reduces to the above $\omega_i = \delta\downarrow 1\downarrow i$.

The next variable type specifies a bit range but not a word range. Unlike the previous two types, this reference can refer to a word variable or an array variable. These two cases are distinguished in the definition below.

(PO3) $PO[(\mathbf{bits}\ I\ \Sigma)]n\rho o\alpha =$
$\quad \delta\in\varnothing\rightarrow \mathbf{err2}(5),$
$\quad \neg\mathbf{legal}(S[\Sigma],\delta\downarrow 2)\rightarrow \mathbf{err2}(5),$
$\quad \delta\in\mathrm{WORD}\rightarrow$
$\quad\quad (\mathbf{size}(S[\Sigma])\ \mathrm{mod}\ n)\neq 0\rightarrow \mathbf{err2}(6),$
$\quad\quad o(\mathbf{aconc}(\alpha,\mathbf{reshape}(n,\mathbf{extract}(\mathbf{norm}(S[\Sigma],\delta\downarrow 2),\delta\downarrow 1))))$
$\quad \delta\in\mathrm{ARRAY}\rightarrow$
$\quad\quad (\mathbf{size}(S[\Sigma])*\mathbf{size}(\delta\downarrow 3)\ \mathrm{mod}\ n)\neq 0\rightarrow \mathbf{err2}(6),$
$\quad\quad o(\mathbf{aconc}(\alpha,\mathbf{reshape}(n,\omega_{p\downarrow 1}\S\omega_{(p\downarrow 1)+1}\S\ \cdots\ \S\omega_{p\downarrow 2})))$
where $\delta = \rho\downarrow 1[I]$
$\quad \omega_i = \mathbf{extract}(\mathbf{norm}(S[\Sigma],\delta\downarrow 2),\delta\downarrow 1\downarrow i)$
$\quad p = \mathbf{norm}(\delta\downarrow 3,\delta\downarrow 3) = \langle 1,\mathbf{size}(\delta\downarrow 3)\rangle$

In the case that the variable is an array variable, the definition is similar to the first definition. The only difference is that instead of using a user-supplied word range, the entire word range is assumed. If the variable is a word variable, then the appropriate locations are extracted from the word location string $\delta\downarrow 1$ and are reshaped and passed to the overlay continuation as usual.

The final variable type specifies neither a bit range nor a word range. Its definition is identical to the previous definition, except that both ranges are defaulted to their full value.

(PO4) $PO[I]n\rho o\alpha =$
$\quad \delta\in\varnothing\rightarrow \mathbf{err2}(5),$
$\quad \neg\mathbf{legal}(S[\Sigma],\delta\downarrow 2)\rightarrow \mathbf{err2}(5),$
$\quad \delta\in\mathrm{WORD}\rightarrow$
$\quad\quad (\mathbf{size}(\delta\downarrow 2)\ \mathrm{mod}\ n)\neq 0\rightarrow \mathbf{err2}(6),$
$\quad\quad o(\mathbf{aconc}(\alpha,\mathbf{reshape}(n,\delta\downarrow 1)))$
$\quad \delta\in\mathrm{ARRAY}\rightarrow$
$\quad\quad (\mathbf{size}(\delta\downarrow 2)*\mathbf{size}(\delta\downarrow 3)\ \mathrm{mod}\ n)\neq 0\rightarrow \mathbf{err2}(6),$
$\quad\quad o(\mathbf{aconc}(\alpha,\mathbf{reshape}(n,\omega_{p\downarrow 1}\S\omega_{p\downarrow 1+1}\S\ \cdots\ \S\omega_{p\downarrow 2})))$
where $\delta = \rho\downarrow 1[I]$
$\quad \omega_i = \delta\downarrow 1\downarrow i$
$\quad p = \mathbf{norm}(\delta\downarrow 3,\delta\downarrow 3) = \langle 1,\mathbf{size}(\delta\downarrow 3)\rangle$

4.3.12 Declarations (D:Dec→X→X, DP:Dec→U→C→C)

The semantics of declarations is among the most complicated in AMDL. There are actually two semantic functions, D and DP. The first, D, can be thought of as the "top-level" semantic function. The second, DP, is required for the semantics of the **proc** declaration. We will consider the top level function first.

The domain for D is the same as for the function P, which we have already examined. Given a binding function q, an environment ρ, a store σ and a declaration continuation χ, it produces an answer by applying the declaration continuation to an updated binding function, environment and store. The simplest type of declaration has the simplest semantics, as follows:

(D3) $D[\![\Pi]\!] = P[\![\Pi]\!](0)$

Here we see that the meaning of the declaration is simply $P[\![\Pi]\!]$ with an argument of "0" which denotes that the variable occurs by itself. This is another example of how currying reduces the size of these equations.

The next type of declaration, the overlay, is not quite as simple. We have already looked at the most complicated part of the semantics, however, in our examination of PO. The definition is

(D2) $D[\![(\textbf{over}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n))]\!]\chi =$
$P[\![\Pi_0]\!](2)\{\lambda q\rho\sigma.PO[\![\Pi_1]\!]n\rho\{PO[\![\Pi_2]\!]n\rho\ \ldots\ \{PO[\![\Pi_n]\!]n\rho o\}\ \ldots\ \}\langle\rangle\}$
where $o = \{\lambda\alpha.\#(\alpha\downarrow 1)\neq \textbf{size}(\delta\downarrow 2)\rightarrow \textbf{err2}(12),$
$\chi q\rho_1\sigma\}$
$n = (\delta\in \text{ARRAY}\rightarrow \textbf{size}(\delta\downarrow 3),$
$1)$
$\delta = \rho\downarrow 1(PI[\![\Pi_0]\!])$
$\rho_1 = \textbf{update}(\rho,1,\rho\downarrow 1[\textbf{update}(\delta.1,(\delta\in \text{ARRAY}\rightarrow\alpha,\alpha\downarrow 1))/PI[\![\Pi_0]\!]])$

First, the variable Π_0 is evaluated with an argument of 2, showing that the variable is being defined in an overlay declaration. The continuation passed to P then applies PO to each of the variables in the defining expression, passing to them the updated ρ and the new variable length n. The length n is determined by examining the updated ρ and examining the variable's newly defined structure. If it is a word variable, the length n is simply 1. If it is an array, then the length is found by taking $\textbf{size}(\delta\downarrow 3)$ where $\delta = \rho\downarrow 1(PI[\![\Pi_0]\!])$.

The concatentation of the array location strings is accomplished by supplying each application of PO with a continuation consisting of an application of PO to the next variable. Note again how the currying has allowed us to perform this concatenation concisely. The partial application of PO provides us with a declaration continuation without the need to form a lambda expresion. The $\langle\rangle$ appearing just inside the rightmost "}" is an empty array location string which is passed to $PO[\![\Pi_1]\!]$ to initialize things.

The declaration continuation o performs some final checking and then applies the original declaration continuation χ to the updated results. The check is that the word size of the resulting

array location string α is equal to the word size of the new variable. This, in addition to the invidual checks occurring in *PO*, assures that the total number of locations required by the new variable equals the total number of locations in the defining expression. The binding function **q** and the store σ passed to the declaration continuation are the same **q** and σ which were passed from $P[\![\Pi_0]\!]$. A new environment ρ_1, however, includes the definition for the new variable. If it is an array variable, then the array location string α is simply used as the defining location string. If it is a word variable, then one layer of structure must be removed from α which is always formed as an array location string. α will contain exactly one location string, in this case, and this one string is referenced simply by $\alpha\downarrow 1$.

We now examine the procedure declaration. The following definition is incomplete. The function *DP* provides the majority of the semantics for procedures.

(D1) $D[\![(\textbf{proc}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n)\ \Delta_1\ \Delta_2\ \ldots\ \Delta_m\ A)]\!]\chi =$
$P[\![\Pi_0]\!](1)\{\lambda q\rho\sigma.\chi(q)(\textit{update}(\rho,2,\rho\downarrow 2[\langle l,n\rangle/PI[\![\Pi_0]\!]]))\sigma[0/l]\}$
where **n** is the subscript in Π_n
$l = \textit{new}(\sigma)$

The operation performed by this function is to declare the identifier, through the application of $P[\![\Pi_0]\!]$, and to then place a new location **l** and the number of procedure arguments **n** in the function component of the environment. The location **l** will later be used as a pointer to the denotation of the procedure body. The number **n** is used as a check to insure that the number of actual parameters matches the number of formal parameters when evaluating **call** expressions. The store σ is updated to reflect the fact that the location **l** is no longer available.

We now come to the function *DP*. As mentioned earlier, this function performs the heart of the operations for defining procedures. The function takes an environment, a store, and an action continuation and produces an answer. The definition is quite involved and we will have to examine it carefully.

(DP1) $DP[\![(\textbf{proc}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n)\ \Delta_1\ \Delta_2\ \ldots\ \Delta_m\ A)]\!]\rho_1\theta_1 =$
$P[\![\Pi_1]\!]0\{P[\![\Pi_2]\!]0\ \ldots\ \{P[\![\Pi_n]\!]0\chi_1\}\}q_1\rho_1\theta_1$
where $q_1 = \lambda I.\oplus$
$\chi_1 = \{D[\![\Delta_1]\!]\{D[\![\Delta_2]\!]\ \ldots\ \{D[\![\Delta_m]\!]\chi_2\}\}\}$
$\chi_2 = \{\lambda q_2\rho_2.DP[\![\Delta_1]\!]\rho_2\{DP[\![\Delta_2]\!]\rho_2\ \ldots\ \{DP[\![\Delta_m]\!]\rho_2\theta_2\}\}\}$
$\theta_2 = \{\lambda s_1.\theta_1(\sigma_1[f/l])\}$
$l = \rho\downarrow 2(PI[\![\Pi_0]\!])\downarrow 1$
$f = \lambda(e_1,e_2,\ \ldots\ ,e_n)\gamma_2.x_1\circ x_2\circ\ \ldots\ \circ x_n\circ\theta_3$
$\theta_3 = A[\![A]\!]\rho_3\gamma_2[\textit{update}(\gamma(l),2,\theta_3)/l]\{\gamma(l)\downarrow 1\}$
$x_i = \textit{setba}(\rho\downarrow 1(PI[\![\Pi_i]\!])\downarrow 1,e_i)$
$\rho_3 = \textit{update}(\rho,4,l)$
and "$\circ$" denotes functional composition

The content of the definition can be summarized as follows:

1. Declare the procedure arguments.
2. Declare the internal definitions.
3. Place the denotation of the procedure action in the store.
4. Apply the action continuation θ to the updated store.

The procedure arguments are evaluated with *P* as if they were simple variable declarations. The binding function $q_1 = \lambda I.\oplus$ is a constant to show that no identifiers have been declared yet. Then the internal declarations are evaluated (shown as χ_1) using the function *D*. Recall that for procedure declarations, *D* simply declares the identifier to be a procedure and establishes its number of arguments and location. *D* does not add to the environment the declarations which are *internal* to a procedure. In this way, after applying *D* to all of the declarations $\Delta_1\ \Delta_2\ ...\ \Delta_m$, the correct environment has been established for the further evaluation of the declarations internal to the Δ's and the evaluation of A locally. This is done by passing the continuation χ_2 to the final $D[\Delta_m]$.

χ_2 applies *DP* itself to each of the Δ's. Note that each call to *DP* uses the same environment ρ_2 as an argument. This makes sense, as the declarations local to (say) Δ_1 do not affect Δ_2. The new stores produced by *DP* are passed along, however, through the nesting of the continuations. The continuation θ_2 is the final step in the process. It applies the original continuation θ_1 to the new store produced by placing the denotation f of A in the store location l.

But what is this f? Well, first of all, it is a function of n expression values. Second, it takes a procedure continuation containing, as you recall, the return addresses or "rest of program"s for previously invoked procedures. The body of f consists of the functional composition of several components. The x_i are store-to-store functions which place the expression values passed in the locations associated with the formal parameters. This indicates that AMDL uses a call-by-value convention for passing of parameters. After this is accomplished, the continuation θ_3 is applied to the resultant store. θ_3 is defined to be *A*[A] applied to the environment ρ_3, an updated procedure continuation γ_2, and an unusual action continuation $\{\gamma_2(l)\downarrow 1\}$.

The environment ρ_3 is just the environment ρ_2 with the last component set to the location l associated with the current procedure. How is the procedure continuation γ_2 updated? Recall that the entries in γ are a triple of jumps, corresponding to the **leave**, **restart** and **resume** operations for the associated procedure. The second (**restart**) component of the entry for the current procedure (pointed to by l) is set to be θ_3 itself. An ensuing **restart**, then, will cause the action A to be re-evaluated. Note that this is not the same as restarting f, for the actual parameters to the procedure are not reassigned to the formal parameters.

The unusual continuation $\{\gamma_2(l)\downarrow 1\}$ is simply the leave component of the current procedure

entry, established by the calling procedure at the time of call. In other words, an implicit leave operation is performed after the last statement in any procedure.

The definition of *DP* for the other two types of declarations are (mercifully) quite simple. Since *DP* is used only for a "second-pass" over procedure declarations, the semantics in the following two instances is to simply pass along the action continuation θ.

(DP2) $DP[\textbf{(over } \Pi_0 \ (\Pi_1 \ \Pi_2 \ \ldots \ \Pi_n))]\rho\theta = \theta$

(DP3) $DP[\Pi_0]\rho\theta = \theta$

4.3.13 Programs (*PROG*:Dec→ANS)

An AMDL program is syntactically just a declaration. (We shall soon see that this is further limited to only procedure declarations.) But what is the meaning of an AMDL program? From the above it would appear that it is just a constant, some member of ANS. This turns out to be quite sensible in that we have included no way to accept any input to the program. Let us look at how this constant is derived.

(PROG1) $PROG[\Delta] =$

$\quad n \neq 0 \rightarrow \textbf{err2}(20),$

$\quad D[\Delta]\{\lambda q\rho.DP[\Delta]\rho\theta\}q_0\rho_0\sigma_0$

where $\Delta = (\textbf{proc } \Pi_0 \ (\Pi_1 \ \Pi_2 \ \ldots \ \Pi_n) \ \Delta_1 \ \Delta_2 \ \ldots \ \Delta_m \ A)$

$\quad n$ is the subscript in Π_n

$\quad q_0 = \{\lambda I.\oplus\}$

$\quad \rho_0 = \langle\{\lambda I.\oplus\},\{\lambda I.\oplus\},\{\lambda I.\oplus\},\oplus\rangle$

$\quad \sigma_0 = \{\lambda I.\oplus\}$

$\quad \theta = \{\lambda\sigma_1.\sigma_1(l_1)(\langle\rangle)\gamma_1\sigma_1\}$

$\quad l_1 = (\rho\downarrow 2[\Pi_0])\downarrow 1$

$\quad \gamma_1 = \gamma_0[\langle\theta_0,\oplus,\oplus\rangle/l_1]$

$\quad \gamma_0 = \{\lambda I.\oplus\}$

$\quad \theta_0 = \{\lambda\sigma.\langle\rangle\}$

We check, first of all, that there are no parameters to the procedure. We then apply the declaration function *D* to the procedure with an initial binding function q_0, environment ρ_0 and store σ_0. The continuation to *D* then applies *DP* to the new environment (now containing the entry for Δ) and a continuation θ. θ extracts the procedure abstraction for Δ by using the environment ρ to determine the location l_1. This l_1 then indexes into σ_1 to obtain the procedure abstraction. We know that this procedure abstraction takes no expression value parameters, so we apply it to the null list $\langle\rangle$. We furthermore apply it to the initial procedure continuation γ_0. This initial procedure continuation contains an initial entry θ_0 for the leave component which defines a null continuation for the procedure. The other two entries are $\oplus$. Finally, the procedure abstraction is applied to the store σ_1 to obtain the program answer.

5.0 Conclusion

Two sets of observations come to mind when considering the foregoing definition, one concerning denotational semantics in general and one concerning the semantics of AMDL.

Looking first at the issue of denotational semantics, we see how denotational semantics has been able to precisely describe even the most complex and tedious details of AMDL. Furthermore, it has done so without reverting to an operational model where unwanted objects such as registers and stacks enter in. On the negative side, the use of Greek characters and short function names greatly reduces the readability of the definition at first glance. I say "first glance," because after working with the equations for a long period of time the reduction in the size of the terms resulting from these conventions seems more important in relation to having mnemonic variables names. Another problem arises in the use of nontrivial data structures, such as we saw with the environment. References such as $\delta\downarrow 3\downarrow 1$ are not easily processed by the human reader. The solutions to both of these problems are obvious, one simply uses mnemonic variable names and record structures with mnemonic selectors. The notation used in this definition is similar to the that used originally in [Milne and Strachey].

Several things that can also be said concerning the semantics of AMDL. First of all, there are several, rather difficult semantic concepts in AMDL, most notably

- Overlays
- Leave, Restart and Resume
- Use of same identifier for structure and procedure

No matter how the semantics of the above features is determined, it is likely to be extremely difficult to understand.

Another problem, referred to earlier, is whether or not to explicitly specify the order of evaluation of constructs. This problem arises due to the possibility of side-effects in AMDL expressions and, in the case of a transfer, the possibility of nondisjoint variables.

The question of run-time errors in a program should also be examined. The current procedure simply says that the first error encountered in program evaluation results in an immediate termination of the program. A method of error recovery might be desirable which, of course, should be rigorously defined.

Finally, it must be mentioned again that parallelism, a fundamental feature of any HDL, has been totally ignored in this definition. The addition of this feature to the definition will have a significant effect. Questions of shared variables, synchronization and process termination will have to be examined. This undoubtedly will increase any conceptual difficulties that currently exist.

All things considered, however, one must be encouraged about the problem of precisely specifying language semantics and, in particular, the semantics of HDLs. Hopefully, this definition will work as a catalyst for further efforts in this area.

6.0 Appendix

6.1 Syntactic Domains

A:Act	Actions
B:Bin	Binary Operators
X:Con	Constants
Z:Dcl	Decode Clauses
Δ:Dec	Declarations
E:Exp	Expressions
I:Ide	Identifiers
N:Num	Numerals
P:Ref	Variable References
K:SRf	Structure References
Σ:Str	Structures
T:Tra	Transfer Operators
ϒ:Una	Unary Operators
Π:Var	Variables

6.2 Syntactic Equations

A ::= **(cond (E A)) | (decode E** Z_1 Z_2 ... Z_n**) | (label I A) | (leave I) | (restart I) | (resume I) | (repeat A) | (seq** A_1 A_2 ... A_n**) | (par** A_1 A_2 ... A_n**) | (call I** E_1 E_2 ... $E_{n\geq 0}$**) | E | (write E)**

B ::= **usplus | usdifference | ustimes | usquotient | usremainder | useql | usneq | uslss | usgtr | usleq | usgeq | ustst | tcplus | tcdifference | tctimes | tcquotient | tcremainder | tceql | tcneq | tclss | tcgtr | tcleq | tcgeq | tctst | ussr0 | ussr1 | ussrr | ussrd | ussl0 | ussl1 | usslr | ussld | usor | usxor | usand | useqv | usconc**

X ::= **(bconst** N_1 N_2**) | (hconst** N_1 N_2**) | (oconst** N_1 N_2**) |** N

Z ::= **(**$(\Sigma_1$ Σ_2 ... $\Sigma_n)$ A**) | (**Σ A**) |** A **| (otherwise** A**)**

Δ ::= **(proc** Π_0 $(\Pi_1$ Π_2 ... $\Pi_{n\geq 0})$ Δ_1 Δ_2 ... $\Delta_{m\geq 0}$ A**) | (over** Π_0 $(\Pi_1$ Π_2 ... $\Pi_n))$ **|** Π

E ::= **(ussub E K) | (B** E_1 E_2**) | (**Υ E**) | (call P** E_1 E_2 ... $E_{n\geq 0}$**) | (T (**P_1 P_2 ... P_n**) E)) |** P **|** X

I ::= The set of legal identifiers

N ::= The set of numerals consisting of the digits 0-9

P ::= **(words (bits I K) E) | (words I E) | (bits I K) |** I

K ::= **(pair** X_1 X_2**) |** E

Σ ::= **(pair** X_1 X_2**) |** X

T ::= **usset | tcset**

Υ ::= **tcminus | usnot**

Π ::= **(words (bits I** Σ_1**)** Σ_2**) | (words I** Σ**) | (bits I** Σ**) |** I

6.3 Semantic Domains

$v:V = \{0,1\}$	Bits
$\beta:B = V^+$	Bit Strings
$e:E = \{\langle m,n\rangle \mid m,n \in N,\ n \geq 0,\ 0 \leq m \leq 2^n-1\}$	Expression Values
$t:T = \{true,false\}$	Booleans
$m,n:N = \{0,1,2,...\}$	Non-negative Integers
$z:Z = \{...,-2,-1,0,1,2,...\}$	Integers
$p:P = N \times N$	Pairs of N
$\varnothing = \{\oplus\}$	Set consisting of the "undeclared" element
$ERR = N$	Error codes
$OUT = N$	Output from **write** action
$ANS = (ERR + OUT)^*$	Final answers
$\sigma:S = L \rightarrow (V + F + \varnothing)$	Stores
$\omega:W = L^+$	Word Location Strings
$\alpha:A = W^+$	Array Location Strings
$w:WORD = W \times P$	Word Variable Descriptors
$a:ARRAY = A \times P \times P$	Array Variable Descriptors
$\theta:C = S \rightarrow ANS$	Action Continuations
$\kappa:K = E \rightarrow C$	Expression Continuations
$\mu:M = P \rightarrow C$	Structure Reference Continuations
$\eta:H = W \rightarrow C$	Variable Reference Continuations
$o:O = A \rightarrow ANS$	Overlay Continuations
$\chi:X = Q \rightarrow U \rightarrow C$	Declaration Continuations
$\gamma:G = L \rightarrow ((J \times (J + \varnothing) \times (J + \varnothing)) + \varnothing)$	Procedure Abstraction Continuations
$\psi:Y = E \rightarrow ANS$	Binary Operator Continuations
$j:J = C$	Jumps
$f:F = E^* \rightarrow G \rightarrow C$	Procedure Abstractions
$q:Q = Ide \rightarrow (\{true\} + \varnothing)$	Local Binding Functions
$\rho:U = UVAR \times UPROC \times ULAB \times L$	Environments
$UVAR = (Ide \rightarrow (ARRAY + WORD + \varnothing))$	
$UPROC = (Ide \rightarrow ((L \times N) + \varnothing))$	
$ULAB = (Ide \rightarrow ((J \times J) + \varnothing))$	

$\delta:D$ = All finite domains which can be constructed from given primitive domains and a finite number of construction operations

6.4 Semantic Functions

6.4.1 Actions (A:Act→U→G→C→C)

(A1) A[(**cond** (E A))]$\rho\gamma\theta = E$[E]$\rho\gamma\{\lambda e.e{\downarrow}1 \neq 0 \rightarrow A$[A]$\rho\gamma\theta, \theta\}$

(A2) A[(**decode** E (Z_1 Z_2 ... Z_n))]$\rho\gamma\theta =$
E[E]$\rho\gamma\{\lambda e.Z$[Z_1](e↓1)(0)$\{Z$[Z_2](e↓1)(1) ...
$\{Z$[Z_n](e↓n)(n − 1)$\{$**err**(16)$\}\rho\gamma\theta\}$ ... $\rho\gamma\theta\}\rho\gamma\theta\}$

(A3) A[(**label** I A)]$\rho\gamma\theta =$
A[A]**update**$(\rho,3,\rho{\downarrow}3[\langle\theta,A$[(**label** I A)]$\rho\gamma\theta\rangle/I])\gamma\theta$

(A4) A[(**leave** I)]$\rho\gamma\theta =$
$\rho{\downarrow}3$[I]$\notin\varnothing \rightarrow \rho{\downarrow}3$[I]↓1,
$\delta\in\varnothing \rightarrow$ **err**(7),
$\gamma(\delta{\downarrow}1)\in\varnothing \rightarrow$ **err**(8),
$\gamma(\delta{\downarrow}1){\downarrow}1$
where $\delta = \rho{\downarrow}2$[I]

(A5) A[(**restart** I)]$\rho\gamma\theta =$
$\rho{\downarrow}3$[I]$\notin\varnothing \rightarrow \rho{\downarrow}3$[I]↓2,
$\delta\in\varnothing \rightarrow$ **err**(7),
$\gamma(\delta{\downarrow}1)\in\varnothing \rightarrow$ **err**(8),
$\gamma(\delta{\downarrow}1){\downarrow}2$
where $\delta = \rho{\downarrow}2$[I]

(A6) A[(**resume** I)]$\rho\gamma\theta =$
$\delta\in\varnothing \rightarrow$ **err**(7),
$\gamma(\delta{\downarrow}1)\in\varnothing \rightarrow$ **err**(8),
$\gamma(\delta{\downarrow}1){\downarrow}3\in\varnothing \rightarrow$ **err**(8),
$\gamma(\delta{\downarrow}1){\downarrow}3$
where $\delta = \rho{\downarrow}2$[I]

(A7) A[(**repeat** A)]$\rho\gamma\theta =$
A[A]$\rho\gamma\{A$[(**repeat** A)]$\rho\gamma\theta\}$

(A8) A[(**seq** A_1 A_2 ... A_n)]$\rho\gamma\theta =$
A[A_1]$\rho\gamma\{A$[A_2]$\rho\gamma$ ... $\{A$[A_n]$\rho\gamma\theta\}$ ... }

(A9) A[(**par** A_1 A_2 ... A_n)]$\rho\gamma\theta =$
A[A_1]$\rho\gamma\{A$[A_2]$\rho\gamma$ ... $\{A$[A_n]$\rho\gamma\theta\}$ ... }

(A10) A[(**call** I E_1 E_2 ... $E_{n\geq 0}$)]$\rho\gamma\theta\sigma_1 =$
$\delta\in\varnothing \rightarrow$ **err2**(9),

$\delta\uparrow 2\#n \rightarrow \textbf{err2}(10)$,

$\neg(\gamma(\delta\downarrow 1)\in\varnothing) \rightarrow \textbf{err2}(17)$,

$E[\![E_1]\!]\rho\gamma\{\lambda e_1 . E[\![E_2]\!]\rho\gamma\{\lambda e_2 . \ \ldots \ E[\![E_n]\!]\rho\gamma$

$\{\lambda e_n . \sigma_1(\delta\downarrow 1)(e_1,e_2,\ldots,e_n)\gamma_1\} \ \ldots \ \}\}\sigma_1$

where $\delta = \rho\downarrow 2(I)$

$j = \theta$

n is the subscript of E_n

$\gamma_1 = \gamma[\textbf{update}(\gamma(\rho\downarrow 4),3,j)/\rho\downarrow 4][\langle j,\oplus,\oplus\rangle/\delta\downarrow 1]$

if $n = 0$, then the last clause in the equation reduces to

$\sigma(\delta\downarrow 1)(\langle\rangle)\gamma_1$

(A11) $A[\![E]\!]\rho\gamma\theta = E[\![E]\!]\rho\gamma\{\lambda e.\theta\}$

(A12) $A[\![(\textbf{write E})]\!]\rho\gamma\theta = E[\![E]\!]\rho\gamma\{\lambda e\sigma.\theta(\sigma)\S\langle e\downarrow 1 \text{ in OUT}\rangle\}$

6.4.2 Binary Operators $(B:\text{Bin}\rightarrow(E\times E)\rightarrow Y\rightarrow\text{ANS})$

(B1) $B[\![\textbf{usplus}]\!](e_1,e_2)\psi = \psi(\langle e_1\downarrow 1 + e_2\downarrow 1, m+1\rangle)$
where $m = \textbf{max}(e_1\downarrow 2, e_2\downarrow 2)$

(B2) $B[\![\textbf{usdifference}]\!](e_1,e_2)\psi = \psi(\textbf{enorm}(e_1\downarrow 1 - e_2\downarrow 1, m+1))$
where $m = \textbf{max}(e_1\downarrow 2, e_2\downarrow 2)$

(B3) $B[\![\textbf{ustimes}]\!](e_1,e_2)\psi = \psi(\langle e_1\downarrow 1 * e_2\downarrow 1, e_1\downarrow 2 + e_1\downarrow 2\rangle)$

(B4) $B[\![\textbf{usquotient}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 = 0 \rightarrow \textbf{err}(15)$,
$\psi(\langle e_1\downarrow 1 / e_2\downarrow 1, e_1\downarrow 1\rangle)$

(B5) $B[\![\textbf{usremainder}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 = 0 \rightarrow \textbf{err}(15)$,
$\psi(\langle e_1\downarrow 1 \bmod e_2\downarrow 1, e_2\downarrow 1\rangle)$

(B6) $B[\![\textbf{useql}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 = e_2\downarrow 1 \rightarrow 1,0),1\rangle)$

(B7) $B[\![\textbf{usneq}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 \neq e_2\downarrow 1 \rightarrow 1,0),1\rangle)$

(B8) $B[\![\textbf{uslss}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 \geq e_2\downarrow 1 \rightarrow 0,1),1\rangle)$

(B9) $B[\![\textbf{usgtr}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 \leq e_2\downarrow 1 \rightarrow 0,1),1\rangle)$

(B10) $B[\![\textbf{usleq}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 \leq e_2\downarrow 1 \rightarrow 1,0),1\rangle)$

(B11) $B[\![\textbf{usgeq}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 \geq e_2\downarrow 1 \rightarrow 1,0),1\rangle)$

(B12) $B[\![\textbf{ustst}]\!](e_1,e_2)\psi = \psi(\langle(e_1\downarrow 1 = e_2\downarrow 1 \rightarrow 1, e_1\downarrow 1 \leq e_2\downarrow 0 \rightarrow 1,2),2\rangle)$

(B13) $B[\![\textbf{tcplus}]\!](e_1,e_2)\psi = \psi(\langle \textbf{tcext}(e_1,m)\downarrow 1 + \textbf{tcext}(e_2,m)\downarrow 1, m+1\rangle)$
where $m = \textbf{max}(e_1\downarrow 2, e_2\downarrow 2)$

(B14) $B[\![\textbf{tcdifference}]\!](e_1,e_2)\psi = \psi(\textbf{enorm}(\textbf{tcext}(e_1,m)\downarrow 1 - \textbf{tcext}(e_2,m)\downarrow 1, m+1))$
where $m = \textbf{max}(e_1\downarrow 2, e_2\downarrow 2)$

(B15) $B[\![\textbf{tctimes}]\!](e_1,e_2)\psi = \psi(\textbf{enorm}(\textbf{tcval}(e_1) * \textbf{tcval}(e_2), e_1\downarrow 2 + e_2\downarrow 2))$

(B16) $B[\![\textbf{tcquotient}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 = 0 \rightarrow \textbf{err}(15)$,

$\psi(\textbf{enorm}(\textbf{tcval}(e_1)/\textbf{tcval}(e_2),e_1\downarrow 2))$

(B17) $B[\![\textbf{tcremainder}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 = 0 \rightarrow \textbf{err}(15),$
$\psi(\textbf{enorm}(\textbf{tcval}(e_1) \bmod \textbf{tcval}(e_2),e_2\downarrow 2))$

(B18) $B[\![\textbf{tceql}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) = \textbf{tcval}(e_2) \rightarrow 1,0),1\rangle)$

(B19) $B[\![\textbf{tcneq}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) \neq \textbf{tcval}(e_2) \rightarrow 1,0),1\rangle)$

(B20) $B[\![\textbf{tclss}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) \geq \textbf{tcval}(e_2) \rightarrow 0,1),1\rangle)$

(B21) $B[\![\textbf{tcgtr}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) \leq \textbf{tcval}(e_2) \rightarrow 0,1),1\rangle)$

(B22) $B[\![\textbf{tcleq}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) \leq \textbf{tcval}(e_2) \rightarrow 1,0),1\rangle)$

(B23) $B[\![\textbf{tcgeq}]\!](e_1,e_2)\psi = \psi(\langle(\textbf{tcval}(e_1) \geq \textbf{tcval}(e_2) \rightarrow 1,0),1\rangle)$

(B24) $B[\![\textbf{tctst}]\!](e_1,e_2)\psi =$
$\psi(\langle(\textbf{tcval}(e_1) = \textbf{tcval}(e_2) \rightarrow 1,\textbf{tcval}(e_1) \leq \textbf{tcval}(e_2) \rightarrow 0,2),2\rangle)$

(B25) $B[\![\textbf{ussr0}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 \geq e_1\downarrow 2 \rightarrow \psi(\langle 0,e_1\downarrow 2\rangle),$
$\psi(\textbf{econv}(\textbf{pwr}(e_2\downarrow 1,0)\S\textbf{extract}(\langle 1,e_1\downarrow 2 - e_2\downarrow 1\rangle,\beta)))$
where $\beta = \textbf{bconv}(e_1)$

(B26) $B[\![\textbf{ussr1}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 \geq e_1\downarrow 2 \rightarrow \psi(\textbf{econv}(\textbf{pwr}(e_1\downarrow 2,1))),$
$\psi(\textbf{econv}(\textbf{pwr}(e_2\downarrow 1,1)\S\textbf{extract}(\langle 1,e_1\downarrow 2 - e_2\downarrow 1\rangle,\beta)))$
where $\beta = \textbf{bconv}(e_1)$

(B27) $B[\![\textbf{ussrr}]\!](e_1,e_2)\psi =$
$n = 0 \rightarrow \psi(e_1),$
$\psi(\textbf{econv}(\textbf{extract}(\langle e_1\downarrow 2 - n + 1,e_1\downarrow 2\rangle,\beta)\S\textbf{extract}(\langle 1,e_1\downarrow 2 - n\rangle,\beta)))$
where $n = e_2\downarrow 1 \bmod e_1\downarrow 2$
$\beta = \textbf{bconv}(e_1)$

(B28) $B[\![\textbf{ussrd}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 \geq e_1\downarrow 2 \rightarrow \psi(\textbf{econv}(\textbf{pwr}(e_1\downarrow 2,\beta\downarrow 1))),$
$\psi(\textbf{econv}(\textbf{pwr}(e_2\downarrow 1,\beta\downarrow 1)\S\textbf{extract}(\langle 1,e_1\downarrow 2 - e_2\downarrow 1\rangle,\beta)))$
where $\beta = \textbf{bconv}(e_1)$

(B29) $B[\![\textbf{ussl0}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 \geq e_1\downarrow 2 \rightarrow \psi(\langle 0,e_1\downarrow 2\rangle),$
$\psi(\textbf{econv}(\textbf{extract}(\langle e_2\downarrow 1 + 1,e_1\downarrow 2\rangle,\beta)\S\textbf{pwr}(e_2\downarrow 1,0)))$
where $\beta = \textbf{bconv}(e_1)$

(B30) $B[\![\textbf{ussl1}]\!](e_1,e_2)\psi =$
$e_2\downarrow 1 \geq e_1\downarrow 2 \rightarrow \psi(\textbf{econv}(\textbf{pwr}(e_1\downarrow 2,1))),$
$\psi(\textbf{econv}(\textbf{extract}(\langle e_2\downarrow 1 + 1,e_1\downarrow 2\rangle,\beta)\S\textbf{pwr}(e_2\downarrow 1,1)))$
where $\beta = \textbf{bconv}(e_1)$

(B31) $B[\![\textbf{sslr}]\!](e_1,e_2)\psi =$
$n = 0 \rightarrow \psi(e_1),$
$\psi(\textbf{econv}(\textbf{extract}(\langle n + 1,e_1\downarrow 2\rangle,\beta)\S\textbf{extract}(\langle 1,n\rangle,\beta)))$
where $n = e_2\downarrow 1 \bmod {}_1\downarrow 2$

β = ***bconv***(e_1)

(B32) B[**ussld**](e_1,e_2)ψ =
$e_2\downarrow1 \geq e_1\downarrow2 \rightarrow$ ***econv***(***pwr***($e_1\downarrow2$,$\beta\downarrow(e_1\downarrow2)$)),
econv(***extract***(⟨$e_2\downarrow1 + 1$,$e_1\downarrow2$⟩,β)§***pwr***($e_2\downarrow1$,$\beta\downarrow(e_1\downarrow2)$))
where β = ***bconv***(e_1)

(B33) B[**usor**](e_1,e_2)ψ = ψ(***econv***(β))
where $\beta\downarrow i$ = ***or***(***adjust***(***bconv***(e_1),m,0)$\downarrow i$,***adjust***(***bconv***(e_2),m,0)$\downarrow i$)
$\#\beta$ = m = ***max***($e_1\downarrow2$,$e_2\downarrow2$)
where ***or***:(V×V)→V
or(v_1,v_2) = ($v_1 + v_2 \geq 1 \rightarrow 1$,0)

(B34) B[**usxor**](e_1,e_2)ψ = ψ(***econv***(β))
where $\beta\downarrow i$ = ***xor***(***adjust***(***bconv***(e_1),m,0)$\downarrow i$,***adjust***(***bconv***(e_2),m,0)$\downarrow i$)
$\#\beta$ = m = ***max***($e_1\downarrow2$,$e_2\downarrow2$)
xor:(V×V)→V
xor(v_1,v_2) = ($v_1 + v_2 = 1 \rightarrow 1$,0)

(B35) B[**usand**](e_1,e_2)ψ = ψ(***econv***(β))
where $\beta\downarrow i$ = ***and***(***adjust***(***bconv***(e_1),m,0)$\downarrow i$,***adjust***(***bconv***(e_2),m,0)$\downarrow i$)
$\#\beta$ = m = ***max***($e_1\downarrow2$,$e_2\downarrow2$)
and:(V×V)→V
and(v_1,v_2) = ($v_1 * v_2$)

(B36) B[**useqv**](e_1,e_2)ψ = ψ(***econv***(β))
where $\beta\downarrow i$ = ***eqv***(***adjust***(***bconv***(e_1),m,0)$\downarrow i$,***adjust***(***bconv***(e_2),m,0)$\downarrow i$)
$\#\beta$ = m = ***max***($e_1\downarrow2$,$e_2\downarrow2$)
eqv:(V×V)→V
eqv(v_1,v_2) = ($v_1 + v_2 \neq 2 \rightarrow 1$,0)

(B37) B[**usconc**](e_1,e_2)ψ = ψ(***econv***(***bconv***(e_1)§***bconv***(e_2)))

6.4.3 Constants (C:Con→E)

(C1) C[(**bconst** N_1 N_2)] = ⟨N[N_1],N[N_2]⟩

(C2) C[(**hconst** N_1 N_2)] = ⟨N[N_1],N[N_2]⟩

(C3) C[(**oconst** N_1 N_2)] = ⟨N[N_1],N[N_2]⟩

6.4.4 Decode Clauses (Z:Dcl→N→N→C→U→G→C)

(Z1) Z[((Σ_1 Σ_2 ... Σ_n) A)]nm$\theta_1\rho\gamma\theta_2$ =
(S[Σ_1]↓1≤n≤S[Σ_1]↓2)∨(S[Σ_2]↓1≤n≤S[Σ_2]↓2)∨ ... ∨
(S[Σ_n]↓1≤n≤S[Σ_n]↓2)→A[A]$\rho\gamma\theta_2$,
θ_1

(Z2) Z[(Σ A)]nm$\theta_1\rho\gamma\theta_2$ =
S[Σ]↓1≤n≤S[Σ]↓2→A[A]$\rho\gamma\theta_2$,
θ_1

(Z3) $Z[\mathrm{A}]\mathrm{nm}\theta_1\rho\gamma\theta_2 =$
$\quad \mathrm{n} = \mathrm{m} \rightarrow A[\mathrm{A}]\rho\gamma\theta_2,$
$\quad \theta_1$

(Z4) $Z[(\textbf{otherwise}\ \mathrm{A})]\mathrm{nm}\theta = A[\mathrm{A}]$

6.4.5 Declarations (D:Dec→X→X, DP:Dec→U→C→C)

(D1) $D[(\textbf{proc}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n)\ \Delta_1\ \Delta_2\ \ldots\ \Delta_m\ \mathrm{A})]\chi =$
$\quad P[\Pi_0](1)\{\lambda q\rho\sigma.\chi(q)(\textbf{update}(\rho,2,\rho\downarrow 2[\langle \mathrm{l},\mathrm{n}\rangle / PI[\Pi_0]]))\sigma[0/\mathrm{l}]\}$
where n is the subscript in Π_n
$\quad \mathrm{l} = \textbf{new}(\sigma)$

(D2) $D[(\textbf{over}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n))]\chi =$
$\quad P[\Pi_0](2)\{\lambda q\rho\sigma.PO[\Pi_1]\mathrm{n}\rho\{PO[\Pi_2]\mathrm{n}\rho\ \ldots\ \{PO[\Pi_n]\mathrm{n}\rho\mathrm{o}\}\ \ldots\ \}\langle\rangle\}$
where $\mathrm{o} = \{\lambda\alpha.\ \#(\alpha\downarrow 1) \neq \textbf{size}(\delta\downarrow 2) \rightarrow \textbf{err2}(6),$
$\qquad\qquad \chi q\rho_1\sigma\}$
$\quad \mathrm{n} = (\delta \in \mathrm{ARRAY} \rightarrow \textbf{size}(\mathrm{x}\downarrow 3),$
$\qquad 1)$
$\quad \delta = \rho\downarrow 1(PI[\Pi_0])$
$\quad \rho_1 = \textbf{update}(\rho,1,\rho\downarrow 1[\textbf{update}(\delta,1,(\delta \in \mathrm{ARRAY} \rightarrow \alpha,\alpha\downarrow 1))/PI[\Pi_0]])$

(D3) $D[\Pi] = P[\Pi](0)$

(DP1) $DP[(\textbf{proc}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n)\ \Delta_1\ \Delta_2\ \ldots\ \Delta_m\ \mathrm{A})]\rho_1\theta_1 =$
$\quad P[\Pi_1]0\{P[\Pi_2]0\ \ldots\ \{P[\Pi_n]0\chi_1\}\}q_1\rho_1\theta_1$
where $q_1 = \lambda \mathrm{I}.\oplus$
$\quad \chi_1 = \{D[\Delta_1]\{D[\Delta_2]\ \ldots\ \{D[\Delta_m]\chi_2\}\}\}$
$\quad \chi_2 = \{\lambda q_2\rho_2.DP[\Delta_1]0\rho_2\{DP[\Delta_2]0\rho_2\ \ldots\ \{DP[\Delta_m]0\rho_2\theta_2\}\}\}$
$\quad \theta_2 = \{\lambda s_1.\theta_1(\sigma_1[\mathrm{f}/\mathrm{l}])\}$
$\quad \mathrm{l} = \rho\downarrow 2(PI[\Pi_0])\downarrow 1$
$\quad \mathrm{f} = \lambda(e_1,e_2,\ \ldots\ ,e_n)\gamma_2.x_1 \circ x_2 \circ\ \ldots\ \circ x_n \circ \theta_3$
$\quad \theta_3 = A[\mathrm{A}]\rho_3\gamma_2[\textbf{update}(\gamma(\mathrm{l}),2,\theta_3)/\mathrm{l}]\{\gamma(\mathrm{l})\downarrow 1\}$
$\quad x_i = \textbf{setba}(\rho\downarrow 1(PI[\Pi_i])\downarrow 1, e_i)$
$\quad \rho_3 = \textbf{update}(\rho_1.4.\mathrm{l})$

(DP2) $DP[(\textbf{over}\ \Pi_0\ (\Pi_1\ \Pi_2\ \ldots\ \Pi_n))]\rho\theta = \theta$

(DP3) $DP[\Pi_0]\rho\theta = \theta$

6.4.6 Expressions (E:Exp→U→G→K→C)

(E1) E[(**ussub** E K)]$\rho\gamma\kappa$ =
E[E]$\rho\gamma$\{λe.k[K]$\rho\gamma\mu$\}
where μ = \{λp.p↓1≥p↓2→κ(***econv***(***extract***(***norm***(p,⟨n − 1,0⟩),β))),
err(14)\},
n = ***max***(p↓1 + 1,e↓2)
β = ***adjust***(***bconv***(e),n,0)

(E2) E[(B E_1 E_2)]$\rho\gamma\kappa$ =
E[E_1]$\rho\gamma$\{λe_1.E[E_2]$\rho\gamma$\{$\lambda e_1\sigma$.B[B](e_1,e_2)\{λe.κ(e)σ\}\}\}

(E3) E[(T E)]$\rho\gamma\kappa$ =
E[E]$\rho\gamma$\{λe.κ(U[T](e))\}\}

(E4) E[(**call** P E_1 E_2 ... $E_{n\geq 0}$)]$\rho\gamma\kappa\sigma_1$ =
$\delta\in\varnothing\rightarrow$***err2***(9),
$\delta\downarrow 2\neq n\rightarrow$***err2***(10),
$\gamma(\delta\downarrow 1)\notin\varnothing\rightarrow$***err2***(17),
E[E_1]$\rho\gamma$\{λe_1.E[E_2]$\rho\gamma$\{λe_2. ... E[E_n]$\rho\gamma$
\{λe_n.$\sigma_1(\delta\downarrow 1)(e_1,e_2,\ldots,e_n)\gamma_1$\} ... \}\}$\sigma_1$
where δ = $\rho\downarrow 2(RI$[P])
γ_1 = γ[***update***($\gamma(\rho\downarrow 4)$,3,j)/$\rho\downarrow 4$][⟨j,⊕,⊕⟩/$\delta\downarrow 1$]
j = R[P]$\rho\gamma$\{$\lambda\omega\sigma_2$.κ(***deref***(ω,σ_2))σ_2\}
n is the subscript of E_n
if n = 0, then the last clause in the equation reduces to
$\sigma(\delta\downarrow 1)(\langle\rangle)\gamma_1$

(E5) E[((P_1 P_1 ... P_n) T E))]$\rho\gamma\kappa$ =
E[E]$\rho\gamma$\{λe.R[P_1]$\rho\gamma$\{$\lambda\omega_1$.R[P_2]$\rho\gamma$\{$\lambda\omega_2$. ... R[P_n]$\rho\gamma\eta$ ... \}\}\}
where η = \{$\lambda\omega_n\sigma$.κe(T[T]((\$\omega_1$\$\omega_2$\$ ... \$\omega_n$),e)$\sigma$)\}

(E6) E[P]$\rho\gamma\kappa$ = R[P]$\rho\gamma$\{$\lambda\omega\sigma$.κ(***deref***(ω,σ))σ\}

(E7) E[X]$\rho\gamma\kappa$ = κ(C[X])

6.4.7 Identifiers

6.4.8 Numbers (N:Num→N)

6.4.9 Programs (*PROG*:Dec→ANS)

(PROG1) *PROG*[Δ] =
n≠0→**err2**(20),
D[Δ]\{λqρ.DP[Δ]$\rho\theta$\}$q_0\rho_0\sigma_0$

where Δ = (**proc** Π_0 (Π_1 Π_2 ... Π_n) Δ_1 Δ_2 ... Δ_m A)

n is the subscript in Π_n

$q_0 = \{\lambda I.\oplus\}$

$\rho_0 = \langle\{\lambda I.\oplus\},\{\lambda I.\oplus\},\{\lambda I.\oplus\},\oplus\rangle$

$\sigma_0 = \{\lambda I.\oplus\}$

$\theta = \{\lambda\sigma_1.\sigma_1(I_1)(\langle\rangle)\gamma_1\sigma_1\}$

$I_1 = (\rho\downarrow 2[\Pi_0])\downarrow 1$

$\gamma_1 = \gamma_0[\langle\theta_0,\oplus,\oplus\rangle/I_1]$

$\gamma_0 = \{\lambda I.\oplus\}$

$\theta_0 = \{\lambda\sigma.\langle\rangle\}$

6.4.10 Variable References (*R*:Ref→U→G→H→C, *RI*:Ref→Ide)

(R1) R[(**words** (**bits** I K) E)]$\rho\gamma\eta$ =

$\delta \notin$ ARRAY→***err***(11),

E[E]$\rho\gamma\{\lambda e.\neg$***legal***$(\langle e\downarrow 1, e\downarrow 1\rangle, \delta\downarrow 3)\rightarrow$***err***(12),

K[K]$\rho\gamma\mu\}$

where $\delta = \rho\downarrow 1$[I]

$\mu = \{\lambda p.\neg$***legal***$(p,\delta\downarrow 2)\rightarrow$***err***(13),

η(***extract***(***norm***$(p,\delta\downarrow 2),\omega)$)$\}$

where ω = ***extract***(***norm***$(\langle e\downarrow 1, e\downarrow 1\rangle, \delta\downarrow 3), \delta\downarrow 1)\downarrow 1$

(R2) R[(**words** I E)]$\rho\gamma\eta$ =

$\delta \notin$ ARRAY→***err***(11),

E[E]$\rho\gamma\{\lambda e.\neg$***legal***$(\langle e\downarrow 1, e\downarrow 1\rangle, \delta\downarrow 3)\rightarrow$***err***(12),

η(***extract***(***norm***$(\langle e\downarrow 1, e\downarrow 1\rangle, \delta\downarrow 3), \delta\downarrow 1)\downarrow 1)\}$

where $\delta = \rho\downarrow 1$[I]

(R3) R[(**bits** I K)]$\rho\gamma\eta$ =

$\delta \notin$ WORD→***err***(11),

K[K]$\rho\gamma\{\lambda p.\neg$***legal***$(p,\delta\downarrow 2)\rightarrow$***err***(13),

η(***extract***(***norm***$(p,\delta\downarrow 2),\delta\downarrow 1)$)$\}$

where $\delta = \rho\downarrow 1$[I]

(R4) R[I]$\rho\gamma\eta$ =

$\delta \notin$ WORD→***err***(11),

$\eta(\delta\downarrow 1)$

where $\delta = \rho\downarrow 1$[I]

(RI1) RI[(**words** (**bits** I K) E)] = I

(RI2) RI[(**words** I E)] = I

(RI3) RI[(**bits** I K)] = I

(RI4) RI[I] = I

6.4.11 Structure References (K:SRf→U→G→M→C)

(K1) $K[\![\textbf{(pair } X_1\ X_2)]\!]\rho\gamma\mu = \mu(C[\![X_1]\!]\downarrow 1, C[\![X_2]\!]\downarrow 1)$

(K2) $K[\![E]\!]\rho\gamma\mu = E[\![E]\!]\rho\gamma\{\lambda e.\mu(e\downarrow 1, e\downarrow 1)\}$

6.4.12 Structures (S:Str→P)

(S1) $S[\![\textbf{(pair } X_1\ X_2)]\!] = \langle C[\![X_1]\!]\downarrow 1, C[\![X_2]\!]\downarrow 1\rangle$

(S2) $S[\![X]\!] = \langle C[\![X]\!]\downarrow 1, C[\![X]\!]\downarrow 1\rangle$

6.4.13 Transfer Operators (T:Tra→(W×E)→S→S)

(T1) $T[\![\textbf{usset}]\!](\omega,e) = \textbf{setba}(\omega, \textbf{bconv}(e))$

(T2) $T[\![\textbf{tcset}]\!](\omega,e) = \textbf{setb}(\omega, \textbf{adjust}(\beta, \#\,\omega, 1))$
where $\beta = \textbf{bconv}(e)$

6.4.14 Variables (P:Var→N→X→X, PO:Var→N→U→O→O)

(P1) $P[\![\textbf{(words (bits I } \Sigma_1)\ \Sigma_2)]\!]n\chi q\rho\sigma_1 =$
$q[I] = \text{true} \rightarrow \textbf{err2}(2)\sigma,$
$\chi q[\text{true}/I]\textbf{update}(\textbf{update}(\rho,1,\rho\downarrow 1[\langle\alpha, S[\![\Sigma_1]\!], S[\![\Sigma_2]\!]\rangle/I]),2,\rho\downarrow 2[\oplus/I])\sigma_2$
where $\alpha = (n = 2 \rightarrow \oplus, \textbf{reshape}(\textbf{size}(S[\![\Sigma_1]\!]),\omega))$
$\sigma_2 = (n = 2 \rightarrow \sigma_1, \textbf{setba}(\omega,\langle 0\rangle)\sigma_1)$
$\omega = \textbf{get}(m,\sigma_1)$
$m = \textbf{size}(S[\![\Sigma_1]\!]) * \textbf{size}(S[\![\Sigma_2]\!])$

(P2) $P[\![\textbf{(words I } \Sigma)]\!]n\chi q\rho = \textbf{err}(3)$

(P3) $P[\![\textbf{(bits I } \Sigma)]\!]n\chi q\rho\sigma_1 =$
$q[I] = \text{true} \rightarrow \textbf{err}(2)\sigma,$
$\chi q[\text{true}/I]\textbf{update}(\textbf{update}(\rho,1,\rho\downarrow 1[\langle\omega, S[\![\Sigma]\!]\rangle/I]),2,\rho\downarrow 2[\oplus/I])\sigma_2$
where $\omega = (n = 2 \rightarrow \oplus, \textbf{get}(n_1,\sigma_1))$
$\sigma_2 = (n = 2 \rightarrow \sigma_1, \textbf{setba}(\omega,\langle 0\rangle)\sigma_1)$
$n_1 = \textbf{size}(S[\![S]\!])$

(P4) $P[\![I]\!]n\chi q\rho =$
$q[I] = \text{true} \rightarrow \textbf{err}(2),$
$n \neq 1 \rightarrow \textbf{err}(3),$
$\chi q[\text{true}/I]\textbf{update}(\textbf{update}(\rho,1,\rho\downarrow 1[\oplus/I]),2,\rho\downarrow 2[\oplus/I])$

(PO1) $PO[\![\textbf{(words (bits I } \Sigma_1)\ \Sigma_2)]\!]n\rho o\alpha =$
$\delta \notin \text{ARRAY} \rightarrow \textbf{err2}(5),$
$\neg(\textbf{legal}(S[\![\Sigma_1]\!],\delta\downarrow 2) \wedge \textbf{legal}(S[\![\Sigma_2]\!],\delta\downarrow 3)) \rightarrow \textbf{err2}(5),$
$(\textbf{size}(S[\![\Sigma_1]\!]) * \textbf{size}(S[\![\Sigma_2]\!]) \bmod n) \neq 0 \rightarrow \textbf{err2}(6),$
$o(\textbf{aconc}(\alpha, \textbf{reshape}(n, \omega_{p\downarrow 1} \S \omega_{p\downarrow 1+1} \S \ \ldots\ \S \omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \textbf{extract}(\textbf{norm}(S[\Sigma_1],\delta\downarrow 2),\delta\downarrow 1\downarrow i)$

$p = \textbf{norm}(S[\Sigma_2],\delta\downarrow 3)$

(PO2) $PO[(\textbf{words}\ I\ \Sigma)]n\rho o\alpha =$

$\delta \notin \text{ARRAY} \rightarrow \textbf{err2}(5),$

$\neg\textbf{legal}(S[\Sigma],\delta\downarrow 3) \rightarrow \textbf{err2}(5),$

$(\textbf{size}(\delta\downarrow 2)*\textbf{size}(S[\Sigma])\ \text{mod}\ n) \neq 0 \rightarrow \textbf{err2}(6),$

$o(\textbf{aconc}(\alpha,\textbf{reshape}(n,\omega_{p\downarrow 1}\S\omega_{p\downarrow 1+1}\S\ \cdots\ \S\omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \delta\downarrow 1\downarrow i$

$p = \textbf{norm}(S[\Sigma],\delta\downarrow 3)$

(PO3) $PO[(\textbf{bits}\ I\ \Sigma)]n\rho o\alpha =$

$\delta \in \varnothing \rightarrow \textbf{err2}(5),$

$\neg\textbf{legal}(S[\Sigma],\delta\downarrow 2) \rightarrow \textbf{err2}(5),$

$\delta \in \text{WORD} \rightarrow$

$(\textbf{size}(S[\Sigma])\ \text{mod}\ n) \neq 0 \rightarrow \textbf{err2}(6),$

$o(\textbf{aconc}(\alpha,\textbf{reshape}(n,\textbf{extract}(\textbf{norm}(S[\Sigma],\delta\uparrow 2),\delta\downarrow 1))))$

$\delta \in \text{ARRAY} \rightarrow$

$(\textbf{size}(S[\Sigma])*\textbf{size}(\delta\uparrow 3)\ \text{mod}\ n) \neq 0 \rightarrow \textbf{err2}(6),$

$o(\textbf{aconc}(\alpha,\textbf{reshape}(n,\omega_{p\downarrow 1}\S\omega_{p\downarrow 1+1}\S\ \cdots\ \S\omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \textbf{extract}(\textbf{norm}(S[\Sigma],\delta\uparrow 2),\delta\downarrow 1\downarrow i)$

$p = \textbf{norm}(\delta\downarrow 3,\delta\downarrow 3) = \langle 1,\textbf{size}(\delta\downarrow 3)\rangle$

(PO4) $PO[I]n\rho o\alpha =$

$\delta \in \varnothing \rightarrow \textbf{err2}(5),$

$\neg\textbf{legal}(S[\Sigma],\delta\downarrow 2) \rightarrow \textbf{err2}(5),$

$\delta \in \text{WORD} \rightarrow$

$(\textbf{size}(\delta\downarrow 2)\ \text{mod}\ n) \neq 0 \rightarrow \textbf{err2}(6),$

$o(\textbf{aconc}(\alpha,\textbf{reshape}(n,\delta\uparrow 1)))$

$\delta \in \text{ARRAY} \rightarrow$

$(\textbf{size}(\delta\downarrow 2)*\textbf{size}(\delta\downarrow 3)\ \text{mod}\ n) \neq 0 \rightarrow \textbf{err2}(6),$

$o(\textbf{aconc}(\alpha,\textbf{reshape}(n,\omega_{p\downarrow 1}\S\omega_{p\downarrow 1+1}\S\ \cdots\ \S\omega_{p\downarrow 2})))$

where $\delta = \rho\downarrow 1[I]$

$\omega_i = \delta\downarrow 1\downarrow i$

$p = \textbf{norm}(\delta\downarrow 3,\delta\downarrow 3) = \langle 1,\textbf{size}(\delta\downarrow 3)\rangle$

6.5 *Support Functions*

(SF1) **abs**:Z→N — Absolute value

abs(z) =
  z≥0→z,
  −z

(SF2) **aconc**:(A×A)→A — Row concatenation of two As

aconc(α_1,α_2) =
  ⟨α_1↓1§α_2↓1,α_1↓2§α_2↓2, ... ,α_1↓(#α_1)§α_2↓(#α_2)⟩
for #α_1 = #α_2

(SF3) **adjust**:(B×N×V)→B — Shortens a bitstring, or extends it on the left with copies of a given element from V

adjust(β,n,v) =
  n≤#β→**extract**(⟨#β − n + 1, #β⟩,β),
  pwr(n − #β,v)§β

(SF4) **bconv**:E→B — Converts an E to a B

bconv(e) = β
such that **econv**(β) = e

(SF4.1) **deref**:(W×S)→E — Dereference function

deref(ω.σ) =
  econv(⟨σ(ω↓1),σ(ω↓2), ..., σ(ω↓(#ω))⟩)
for #ω≥1, σ(ω↓i)∈V, i = 1,2,...,#ω

(SF5) **econv**:B→E — Converts a B to an E

econv(β) = ⟨n, #β⟩
where n = (#β = 1→β↓1,
  β↓(#β) + 2*(e↓1))
  e = **econv**(**extract**(⟨1, #β − 1⟩, #β))

(SF6) **enorm**:(Z×N)→E — Normalizes ⟨Z,N⟩ to an E

enorm(z,n) =
  z≥0→⟨z mod 2^n,n⟩,
  enorm(2^n + z,n)
for n≥1

(SF7) **err**:N→S→ANS — Program Error

err(n)σ = ⟨n in ERR⟩

(SF7a) **err2**:N→ANS — Program Error

err2(n) = ⟨n in ERR⟩

(SF8) **extract**:(P×D^+)→D^+ — Returns subsequence of member of D^+

extract(p,δ) =

$\langle \delta\downarrow(p\downarrow 1), \delta\downarrow((p\downarrow 1)+1), \ \ldots \ , \delta\downarrow(p\downarrow 2)\rangle$
for $1 \leq p\downarrow 1 \leq p\downarrow 2 \leq \# \delta$

(SF9) ***get***:$(N \times S) \rightarrow L$ Gets locations from an S

get(n,σ) =
$n = 1 \rightarrow \langle$***new***$(\sigma)\rangle$,
$\langle$***new***$(\sigma)\rangle$ §***get***$(n\text{-}1, \sigma[0/$***new***$(\sigma)])$
for $n \geq 1$

(SF10) ***legal***:$(P \times P) \rightarrow T$ Determines if access is legal

legal(p_1,p_2) =
$p_2\downarrow 1 \leq p_2\downarrow 2 \rightarrow (p_2\downarrow 1 \leq p_1\downarrow 1 \leq p_1\downarrow 2 \leq p_2\downarrow 2)$,
$(p_2\downarrow 1 \geq p_1\downarrow 1 \geq p_1\downarrow 2 \geq p_2\downarrow 2)$

(SF11) ***max***:$(N \times N) \rightarrow N$ Maximum

max(n_1,n_2) =
$n_1 \leq n_2 \rightarrow n_2$,
n_1

(SF12) ***new***:$S \rightarrow L$ Gets a sequence of locations from an S

new(σ) =
l such that $\sigma(l) = \oplus$

(SF13) ***norm***:$(P \times P) \rightarrow P$ Normalizes access

norm(p_1,p_2) =
$p_2\downarrow 1 \leq p_2\downarrow 2 \rightarrow \langle p_1\downarrow 1 - p_2\downarrow 1 + 1, p_1\downarrow 2 - p_2\downarrow 1 + 1\rangle$,
$\langle p_2\downarrow 1 - p_1\downarrow 1 + 1, p_2\downarrow 1 - p_1\downarrow 2 + 1\rangle$
for ***legal***(p_1,p_2)

(SF14) ***pwr***:$(N \times V) \rightarrow B$ Generates tuples of an S

pwr$(n,v) = \langle v_1, v_2, \ \ldots \ , v_n\rangle$
where $v_i = v, \ i = \{1,2, \ \ldots \ ,n\}$
for $n \geq 1$

(SF15) ***ravel***:$A \rightarrow W$ Concatenates rows of A

ravel$(\alpha) = \alpha\downarrow 1 \S \alpha\downarrow 1 \S \ \ldots \ \S \alpha\downarrow(\# \alpha)$

(SF16) ***reshape***:$(N \times W) \rightarrow A$ Reshapes a W into an A

reshape(n,ω) =
$\langle\langle \alpha\downarrow 1, \alpha\downarrow 2, \ \ldots \ , \alpha\downarrow m\rangle$,
$\langle \omega\downarrow(m+1), \omega\downarrow(m+1), \ \ldots \ , \omega\downarrow(2*m)\rangle$,
.....
$\langle \omega\downarrow((n-1)*m+1), \omega\downarrow((n-1)*m+2), \ \ldots \ , \omega\downarrow(n*m)\rangle\rangle$
where $m = (\# \omega)/n$
for $n \geq 1$ and $(\# \omega \bmod n) = 0$

(SF17) ***setb***:(W×B)→S→S — Updates S with a B

setb$(\omega,\beta)\sigma = \sigma[\beta\downarrow 1/\omega\downarrow 1][\beta\downarrow 2/\omega\downarrow 2]$ ... $[\beta\downarrow n/\omega\downarrow n]$

for $\#\omega = \#\beta$

(SF18) ***setba***:(W×B)→S→S — Updates S with *adjusted* B

setba(ω,β) = ***setb***$(\omega,$***adjust***$(\beta, \#\omega, 0))$

(SF19) ***size***:P→N — Gives "size" of a P

size(p) = ***abs***(p↓1 − p↓2) + 1

(SF20) ***tcext***:(E×N)→N — Extends E and returns "value" part

tcext(e,n) = ***econv***(***adjust***$(\beta$,n,$\beta\downarrow 1))\downarrow 1$

where β = ***bconv***(e)

for $n \geq 1$

(SF21) ***tcval***:E→Z — Two's complement interpretation of E

tcval(e) =

$e\downarrow 1 \geq 2^{n-1} \rightarrow e\downarrow 1 - 2^n$,

$e\downarrow 1$

where n = e↓2

(SF22) ***update***:$(D^+ \times N \times D) \rightarrow D^+$ — List update function

update(δ,n,d) =

$\langle \delta\downarrow 1, \delta\downarrow 2, \ \ldots\ , \delta\downarrow(n-1), d, \delta\downarrow(n+1), \ \ldots\ \delta\downarrow(\#\delta) \rangle$

for $1 \leq n \leq \#\delta$

6.6 Error Codes

1) Program defined with formal parameters
2) Multiply defined identifier
3) Variable declared without bit structure
4) Undefined variable reference in overlay
5) Illegal word or array reference in overlay
6) Variable reference does not map evenly in overlay
7) **leave**, **restart** or **resume** of undefined procedure or label
8) **leave**, **restart** or **resume** of inactive procedure
9) Reference to undefined procedure
10) Number of actual parameters does not match number of formal parameters in procedure call
11) Undefined or improper variable reference
12) Illegal word specified in array reference
13) Illegal bit sequence specified in word reference
14) Illegal bit sequence specified in **ussub** expression
15) Division by zero
16) Unsatisfied **decode**
17) Recursive procedure call

7.0 Bibliography

Barbacci, M.R., Barnes, G.E., Cattell, R.G. and Siewiorek, D.P., 1977, "The ISPS Computer Description Language," Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Bell, C.G. and Newell, A., 1979, *Computer Structures: Readings and Examples*, McGraw-Hill Book Company, Inc., New York.

Evangelisti, C.J., Goertzel, G. and Ofek, H., 1976, "LCD - Language for Computer Design(Revised), IBM Research Report RC 6244, IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

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

Marcotty, M., Ledgard, H.F. and Bochmann, G.V., 1976, "A Sampler of Formal Definitions," *Computing Surveys*, Vol. 8, No. 2, pp. 191-276.

McCarthy, J., 1966, "A Formal description of a subset of Algol, *Formal Language Description Languages for Computer Programming* (ed. Steel, T.B.), North-Holland, Amsterdam, pp. 1-12.

Milne, R. and Strachey, C., 1976, *A Theory of Programming Language Semantics*, Chapman and Hall, London.

Scott, D.S., 1976, "Data Types as Lattices," *Proceedings of the 1974 Colloquium in Mathematical Logic, Kiel*, Springer-Verlag, Berlin.

Teitelman, W., 1975, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California.

TRW Defense and Space Systems Group, 1977, *Smite Reference Manual*, TR-77-364, Rome Air Development Center, Rome, New York.