The nanoAVA Definition

Dan Craigen, Mark Saaltink, Michael K. Smith

Technical Report 21 June, 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponored in part by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic, Inc.

Abstract

This document comprises a complete description (formal and informal) of the nanoAVA subset of Ada. As such, it represents our first attempt at a complete formal and informal description of AVA (A Verifiable Ada). The intent of this work was twofold. First, we wanted to set a least upper bound. We wanted to be sure that we could formally specify an *extremely* trivial subset of Ada before embarking on a more ambitious subset. Secondly we wanted to experiment with the technical approach to the definition and its presentation to the reader.

Two definitional techniques were experimented with. Included in this report is

1. A denotational definition of nanoAVA and a Boyer-Moore translation of it.
2. A Lisp based definition of nanoAVA.

This reflects the past experience of the project members. Smith originally did a somewhat clumsy Lisp description of the static and dynamic semantics. Saaltink responded with a very clean denotational definition which fed back into the Lisp definition presented here. We have done proofs using the Boyer Moore description derived from the denotational definition and have executed the Lisp version.

This document consists of three primary parts.

1. nanoAVA Language Reference Manual: This informal description was produced largely by subsetting the Ada Language Reference Manual [DoD 83].
2. Denotational Definition:
    a. The Static and Dynamic Semantics of nanoAVA: This very brief section demonstrates the compactness and readability of the denotational approach.
    b. Following this section is a translation to Boyer Moore with the statement of some associated proofs.
3. Lisp Definition: The elements of the Lisp definition have been tied together so that, in conjunction with a lexical scanner (again in Lisp), we can parse in a nanoAVA program, check its semantics, and interpret it.
    a. Syntax of nanoAVA: The grammar presented accepts a strict subset of Ada. Associated with each rule is a form that describes how an internal representation (the input to static semantic analysis) is constructed.
    b. Static Semantics of nanoAVA: This section describes the semantic constraints that are placed on the output of the syntactic component. Forms are analyzed in the context of an *environment*, which contains descriptors for defined objects (predefined types, user defined types, routine definitions, etc.). Forms that are accepted semantically are converted to the internal form expected by the interpreter definition. Some of these converted forms augment the environment is specified ways.
    c. Dynamic Semantics of nanoAVA: The dynamic semantics of the language are described operationally, via an interpreter definition.

PART I: INTRODUCTION

AVA (A Verifiable Ada) is intended to be a formally defined subset of the Ada programming language that is large enough to support simple, standalone applications.

This document comprises a complete description (formal and informal) of the nanoAVA subset of Ada. As such, it represents our first attempt at a complete formal and informal description of AVA. The intent of this work was twofold. First, we wanted to set a least upper bound. We wanted to be sure that we could formally specify an *extremely* trivial subset of Ada before embarking on a more ambitious subset. Secondly we wanted to experiment with the technical approach to the definition and its presentation to the reader.

In some cases these goals conflict. That is, nanoAVA does not require a number of features that we are relatively certain will be needed for the AVA definition. At some points in this document we have included mechanisms more complex than strictly required for nanoAVA due to an inability to refrain from looking ahead to AVA.

The prototypical nanoAVA program is the `swap` procedure below.

```
procedure swap (x, y: in out INTEGER) is
  temp: constant INTEGER := x;
 begin
  x := y;
  y := temp;
 end ;
```

A nanoAVA program consists of a single procedure definition. The only data types are INTEGER and BOOLEAN. There are no control structures other than BEGIN - END statement sequencing. The only statement is assignment. Expressions are of the form IDENTIFIER or IDENTIFIER OPERATOR IDENTIFIER, where the operator must be one of the relational operators, "=", "<", ">", "<=", and ">=".

We have attempted to thoroughly cross-reference this document. In addition to the index there are pointers in the text between the various definitional components. For example, where the null statement is defined in the reference manual part there are pointers to the pages where the syntax and the denotational, static and dynamic semantics are defined.

```
    null_statement ::= null;                                    [Deno: 52,Syntax: 69,Static: 77,Dynamic: 81]
```

Similarly, in the semantic definitions you will see a pointer back to the definition of the language element in the Reference Manual part of the document.

PART II: LANGUAGE REFERENCE MANUAL

As modified by
Dan Craigen and Mark Saaltink

The numbering of sections in this part of the nanoAVA description are taken from the full Ada Language reference Manual for ease of reference.

Chapter II-1
INTRODUCTION

n-AVA (nanoAVA) is a subset of Ada.

II-1.1 Scope of the Standard

This standard specifies the form and meaning of program units written in n-AVA. Its purpose is to promote the portability of n-AVA programs to a variety of data processing systems.

II-1.1.1 Extent of the Standard

This standard specifies:

(a) The form of a program unit written in n-AVA.

(b) The effect of translating and executing such a program unit.

(c) The manner in which program units may be combined to form n-AVA programs.

(d) The predefined program units that a conforming implementation must supply.

(e) The permissible variations within the standard, and the manner in which they must be specified.

(f) Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program unit containing such violations.

(g) Those violations of the standard that a conforming implementation is not required to detect.

This standard does not specify:

(h) The means whereby a program unit written in n-AVA is transformed into object code executable by a processor.

(i) The means whereby translation or execution of program units is invoked and the executing units are controlled.

(j) The size or speed of the object code, or the relative execution speed of different language constructs.

(k) The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages.

(l) The effect of executing a program unit that contains any violation that a conforming implementation is not required to detect.

(m) The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

Where this standard specifies that a program unit written in n-AVA has an exact effect, this effect is the operational meaning of the program unit and must be produced by all conforming implementations. Where this standard specifies permissible variations in the effects of constituents of a program unit written in n-AVA, the operational meaning of the program unit as a whole is understood to be the range of possible effects that result from all these variations, and a conforming implementation is allowed to produce any of these possible effects.

II-1.1.2 Conformity of an Implementation With the Standard

A conforming implementation is one that:

(a) Correctly translates and executes legal program units written in n-AVA, provided that they are not so large as to exceed the capacity of the implementation.

(b) Rejects all program units that are so large as to exceed the capacity of the implementation.

(c) Rejects all program units that contain errors whose detection is required by the standard.

(d) Supplies all predefined program units required by the standard.

(e) Contains no variations except where the standard permits.

(f) Specifies all such permitted variations in the manner prescribed by the standard.

II-1.2 Structure of the Standard

This reference manual contains various chapters, annexes, and appendices. Each is numbered to correspond to a section of the Ada Reference Manual.

Each chapter is divided into sections that have a common structure. Each section introduces its subject, gives any necessary syntax rules, and describes the semantics of the corresponding language constructs. Examples and notes, and then references, may appear at the end of a section.

Examples are meant to illustrate the possible forms of the constructs described. Notes are meant to emphasize consequences of the rules described in the section or elsewhere. References are meant to attract the attention of readers to a term or phrase having a technical meaning defined in another section.

The standard definition of the n-AVA programming language consists of the chapters and the annexes, subject to the following restriction: the material in each of the items listed below is informative, and not part of the standard definition of the n-AVA programming language:

- Section 1.4 Language summary

- The examples, notes, and references given at the end of each section

- Each section whose title starts with the word "Example" or "Examples"

II-1.3 Design Goals and Sources: Removed

II-1.4 Language Summary

An n-AVA program is composed of one program unit. Program units are subprograms (which define executable algorithms).

Program Units

A subprogram is the basic unit for expressing an algorithm. There is one kind of subprogram: procedures. A procedure is the means of invoking a series of actions. For example, it may read data, update variables, or produce some output. It may have parameters, to provide a controlled means of passing information between the procedure and the point of call.

Declarations and Statements

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

The declarative part associates names with declared entities. For example, a name may denote a constant or a variable.

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession.

An assignment statement changes the value of a variable.

Data Types

Every object in the language has a type, which characterizes a set of values and a set of applicable operations.

An enumeration type defines an ordered set of distinct enumeration literals. The enumeration type BOOLEAN is predefined.

Numeric types provide a means of performing numerical computations. Exact computations use integer types, which denote sets of consecutive integers. The numeric type INTEGER is predefined.

II-1.5 Method of Description and Syntax Notation

The form of n-AVA program units is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules.

The meaning of n-AVA program units is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. This narrative employs technical terms whose precise definition is given in the text (references to the section containing the definition of a technical term appear at the end of each section that uses the term).

All other terms are in the English language and bear their natural meaning, as defined in Webster's Third New International Dictionary of the English Language.

The context-free syntax of the language is described using a simple variant of Backus-Naur-Form. In particular,

(a) Lower case words, some containing embedded underlines, are used to denote syntactic categories, for example:

```
adding_operator
```

Whenever the name of a syntactic category is used apart from the syntax rules themselves, spaces take the place of the underlines (thus: adding operator).

(b) Boldface words are used to denote reserved words, for example:

```
array
```

(c) Square brackets enclose optional items. Thus the two following rules are equivalent.

```
return_statement ::= return [expression];
return_statement ::= return; | return expression;
```

(d) Braces enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent.

```
term ::= factor {multiplying_operator factor}
term ::= factor | term multiplying_operator factor
```

(e) A vertical bar separates alternative items unless it occurs immediately after an opening brace, in which case it stands for itself:

```
letter_or_digit ::= letter | digit
component_association ::= [choice {| choice} =>] expression
```

(f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example type_name and task_name are both equivalent to name alone.

II-1.6 Classification of Errors

The language definition classifies errors into a single category:

(a) Errors that must be detected at compilation time by every n-AVA compiler.

These errors correspond to any violation of a rule given in this reference manual. In particular, violation of any rule that uses the terms must, allowed, legal, or illegal belongs to this category. Any program that contains such an error is not a legal n-AVA program; on the other hand, the fact that a program is legal does not mean, per se, that the program is free from other forms of error.

[We did not include (b) because it would introduce exceptions. Our section 10.1 allows for abandonment of programs.]

Chapter II-2
LEXICAL ELEMENTS

The text of a program consists of the text of a compilation. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this chapter.

References: character 2.1, compilation 10.1, lexical element 2.2

II-2.1 Character Set

The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO seven-bit coded character set (ISO standard 646), and is represented (visually) by a graphical symbol. Some graphic characters are represented by different graphical symbols in alternative national representations of the ISO character set. The description of the language definition in this standard reference manual uses the ASCII graphical symbols, the ANSI graphical representation of the ISO character set.

```
graphic_character ::= basic_graphic_character    | lower_case_letter | other_special_character

basic_graphic_character ::=     upper_case_letter | digit
  | special_character | space_character

basic_character ::=     basic_graphic_character | format_effector
```

The basic character set is sufficient for writing any program. The characters included in each of the categories of basic graphic characters are defined as follows:

(a) upper case letters A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) digits 0 1 2 3 4 5 6 7 8 9

(c) special characters " # & ’ ( ) * + , - . / : ; < = > _ |

(d) the space character

Format effectors are the ISO (and ASCII) characters called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

The characters included in each of the remaining categories of graphic characters are defined as follows:

(e) lower case letters a b c d e f g h i j k l m n o p q r s t u v w x y z

(f) other special characters ! $ % ? @ [\ ] ^ ‘ { }

Notes:

The ISO character that corresponds to the sharp graphical symbol in the ASCII representation appears as a pound sterling symbol in the French, German, and United Kingdom standard national representations. In any case, the font design of graphical symbols (for example, whether they are in italic or bold typeface) is not part of the ISO standard.

The meanings of the acronyms used in this section are as follows: ANSI stands for American National Standards Institute, ASCII stands for American Standard Code for Information Interchange, and ISO stands for International Organization for Standardization.

The following names are used when referring to special characters and other special characters:

symbol	name	symbol	name
"	quotation	>	greater than
#	sharp	_	underline
&	ampersand	|	vertical bar
’	apostrophe	!	exclamation mark
(	left parenthesis	$	dollar
)	right parenthesis	%	percent
*	star, multiply	?	question mark
+	plus	@	commercial at
,	comma	[	left square bracket
-	hyphen, minus	\	back-slash
.	dot, point, period	]	right square bracket
/	slash, divide	^	circumflex
:	colon	‘	grave accent
;	semicolon	{	left brace
<	less than	}	right brace
=	equal	‘	tilde

II-2.2 Lexical Elements, Separators, and Delimiters

The text of a program consists of the text of a compilation. The text of each compilation is a sequence of separate lexical elements.

Each lexical element is either a delimiter, an identifier (which may be a reserved word), or a comment. The effect of a program depends only on the particular sequences of lexical elements that form its compilation, excluding the comments, if any.

In some cases an explicit separator is required to separate adjacent lexical elements (namely, when without separation, interpretation as a single lexical element is possible). A separator is any of a space character, a format effector, or the end of a line. A space character is a separator except within a comment. Format effectors other than horizontal tabulation are always separators. Horizontal tabulation is a separator except within a comment.

The end of a line is always a separator. The language does not define what causes the end of a line. However if, for a given implementation, the end of a line is signified by one or more characters, then these characters must be format effectors other than horizontal tabulation. In any case, a sequence of one or more format effectors other than horizontal tabulation must cause at least one end of line.

One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier or a numeric literal and an adjacent identifier or numeric literal.

A delimiter is either one of the following special characters (in the basic character set)

```
& ' ( ) * + , - . / : ; < = > |
```

or one of the following compound delimiters each composed of two adjacent special characters

```
=>   ..   **   :=   /=   >=   <=   <<   >>   <>
```

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string literal, character literal, or numeric literal.

The remaining forms of lexical element are described in other sections of this chapter.

Notes:

Each lexical element must fit on one line, since the end of a line is a separator. The quotation, sharp, and underline characters, likewise two adjacent hyphens, are not delimiters, but may form part of other lexical elements.

The following names are used when referring to compound delimiters:

delimiter	name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: "becomes")
/=	inequality (pronounced: "not equal")
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

References: comment 2.7, compilation 10.1, format effector 2.1, identifier 2.3, reserved word 2.9, space character 2.1, special character 2.1

II-2.3 Identifiers

Identifiers are used as names and also as reserved words.

```
identifier ::=      letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit
letter ::= upper_case_letter | lower_case_letter
```

All characters of an identifier are significant, including any underline character inserted between a letter or digit and an adjacent letter or digit. Identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

Examples:

```
COUNT     X    get_symbol   Ethelyn   Marion

SNOBOL_4   X1   PageCount    STORE_NEXT_ITEM
```

Note:

No space is allowed within an identifier since a space is a separator.

References: digit 2.1, lower case letter 2.1, name 4.1, reserved word 2.9, separator 2.2, space character 2.1, upper case letter 2.1

II-2.4 Numerica Literals: Removed

II-2.5 Character Literals: Removed

II-2.6 String Literals: Removed

II-2.7 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment can appear on any line of a program. The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the effect of a program; their sole purpose is the enlightenment of the human reader.

Examples:

```
--  the last sentence above echoes the Algol 68 report

end;  --  processing of LINE is complete

--  a long comment may be split onto
--  two or more consecutive lines

----------------  the first two hyphens start the comment
```

Note:

Horizontal tabulation can be used in comments, after the double hyphen, and is equivalent to one or more spaces (see 2.2).

References: end of a line 2.2, illegal 1.6, legal 1.6, space character 2.1

II-2.8 Pragmas

II-2.9 Reserved Words

The identifiers listed below are called reserved words and are reserved for special significance in the language. For readability of this manual, the reserved words appear in lower case boldface.

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	

all	do	in		task
and		is	package	terminate
array			pragma	then
at	else		private	type
	elsif	limited	procedure	
	end	loop		
begin	entry		raise	use
body	exception		range	
	exit	mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

A reserved word must not be used as a declared identifier.

Notes:

Reserved words differing only in the use of corresponding upper and lower case letters are considered as the same (see 2.3).

References: declaration 3.1, identifier 2.3, lower case letter 2.1, upper case letter 2.1

II-2.10 Allowable Replacements of Characters: Removed

Chapter II-3

DECLARATIONS AND TYPES

This chapter describes the types in the language and the rules for declaring constants and variables.

II-3.1 Declarations

The language defines several kinds of entities that are declared, either explicitly or implicitly, by declarations. Such an entity can be an object, a type, a subprogram, a formal parameter (of a subprogram), or an operation (see 3.3.3).

There are several forms of declaration. A basic declaration is either an object declaration, a type declaration, or a subprogram body. [How do we handle predefined operators? Are they subprograms?]

Certain forms of declaration always occur (explicitly) as part of another declaration; these forms are parameter specifications. [This is a prevarication. LEFT and RIGHT parameters may be implicitly declared. See ARM 4.5.]

The remaining forms of declaration are implicit. Certain operations are implicitly declared (see 3.3.3).

For each form of declaration the language rules define a certain region of text called the scope of the declaration (see 8.2). Several forms of declaration associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the associated declared entity; these places are defined by the visibility rules (see 8.3). At such places the identifier is said to be a name of the entity (its simple name); the name is said to denote the associated entity.

Certain forms of declaration associate some notation with an explicitly or implicitly declared operation.

The process by which a declaration achieves its effect is called the elaboration of the declaration; this process happens during program execution.

After its elaboration, a declaration is said to be elaborated. Prior to the completion of its elaboration (including before the elaboration), the declaration is not yet elaborated. The elaboration of any declaration has always at least the effect of achieving this change of state (from not yet elaborated to elaborated). The phrase "the elaboration has no other effect" is used in this manual whenever this change of state is the only effect of elaboration for some form of declaration. An elaboration process is also defined for declarative parts, declarative items, and compilation units (see 3.9 and 10.5).

Note:

The syntax rules use the term identifier for the first occurrence of an identifier in some form of declaration; the term simple name is used for any occurrence of an identifier that already denotes some declared entity.

References: declarative item 3.9, declarative part 3.9, elaboration 3.9, identifier 2.3, name 4.1, object declaration 3.2.1, operation 3.3, operator symbol 6.1, parameter specification 6.1, scope 8.2, simple name 4.1, subprogram body 6.3, subprogram specification 6.1, visibility 8.3

II-3.2 Objects

An object is an entity that contains (has) a value of a given type. An object is one of the following:

- an object declared by an object declaration, or
- a formal parameter of a subprogram.

```
object_declaration  ::=        identifier_list : [constant] type_mark := expression;

identifier_list ::=  identifier {, identifier}                                   [Syntax: 68]
```

An object declaration is called a single object declaration if its identifier list has a single identifier; it is called a multiple object declaration if the identifier list has two or more identifiers. A multiple object declaration is equivalent to a sequence of the corresponding number of single object declarations. For each identifier of the list, the equivalent sequence has a single object declaration formed by this identifier, followed by a colon and by whatever appears at the right of the colon in the multiple object declaration; the equivalent sequence is in the same order as the identifier list.

A similar equivalence applies also for the identifier lists of parameter specifications.

In the remainder of this reference manual, explanations are given for declarations with a single identifier; the corresponding explanations for declarations with several identifiers follow from the equivalence stated above.

Example:

```
--  the multiple object declaration

JOHN, PAUL : INTEGER := GEORGE;

--  is equivalent to the two single object declarations in the order given

JOHN : INTEGER := GEORGE;
PAUL : INTEGER := GEORGE;
```

References: declaration 3.1, expression 4.4, formal parameter 6.1, identifier 2.3, parameter specification 6.1, scope 8.2, simple name 4.1, subprogram 6, type 3.3, type_mark 3.3.2

II-3.2.1 Object Declarations

An object declaration declares an object whose type is given by a type mark. The expression specifies an initial value for the declared object; the type of the expression must be that of the object.

The declared object is a constant if the reserved word constant appears in the object declaration. The value of a constant cannot be modified after initialization.

An object that is not a constant is called a variable (in particular, the object declared by an object declaration that does not include the reserved word constant is a variable). The only way to change the value of a variable is directly by an assignment.

The elaboration of an object declaration proceeds as follows:

(b) The initial value is obtained by evaluating the corresponding
expression.

(c) The object is created.

(d) The initial value is assigned to the object.

The steps (b) to (d) are performed in the order indicated.

Examples of variable declarations:

```
   COUNT, SUM  : INTEGER := ZERO;
```

Examples of constant declarations:

```
   LIMIT     : constant INTEGER := SUM;
```

References: assignment 5.2, declaration 3.1, elaboration 3.9, evaluation 4.5, expression 4.4, formal parameter 6.1, reserved word 2.9, subprogram 6, type 3.3, type mark 3.3.2

II-3.2.2 Number Declarations: Removed

II-3.3 Types

A type is characterized by a set of values and a set of operations.

There exists a single class of types. Scalar types are integer types and types defined by enumeration of their values.

The name of a class of types is used in this manual as a qualifier for objects and values that have a type of the class considered. For example, the term "integer object" is used for an object whose type is an integer type.

References: integer type 3.5.4, object 3.2.1, operation 3.3.3

[We omitted 3.3.1 since type declarations are implicit in n-AVA. We are drawing an analogy with the definition of literals within package STANDARD -- in full Ada.]

II-3.3.1 Type Declarations: Removed

II-3.3.2 Subtype Declarations

```
  type_mark ::= type_name                    [ARM: 21,Static: 76,Dynamic: 81]
```

A type mark denotes a type.

References: name 4.1

II-3.3.3 Classification of Operations

The set of operations of a type includes the explicitly declared subprograms that have a parameter or result of the type; such subprograms are necessarily declared after the type declaration.

The remaining operations are each implicitly declared for a given type declaration, immediately after the type definition. These implicitly declared operations comprise the basic operations and the predefined operators.

A basic operation is an operation that is inherent in one of the following:

- An assignment (in assignment statements and initializations)

References: assignment 5.2, formal parameter 6.1, initial value 3.2.1, subprogram 6, type 3.3

Note:

Assignment is an operation that operates on an object and a value.

II-3.4 Derived Types: Removed

II-3.5 Scalar Types

Scalar types comprise enumeration types and integer types. Enumeration types and integer types are called discrete types. Integer types are called numeric types. All scalar types are ordered, that is, all relational operators are predefined for their values.

References: integer type 3.5.4, relational operator 4.5 4.5.2

II-3.5.1 Enumeration Types

II-3.5.2 Character Types

II-3.5.3 Boolean Types

There is a predefined enumeration type named BOOLEAN. It contains the two literals FALSE and TRUE ordered with the relation FALSE < TRUE.

References: type 3.3

II-3.5.4 Integer Types

The predefined integer type is the type INTEGER. The range of this type must be symmetric about zero, excepting an extra negative value which may exist in some implementations.

References: type 3.3

II-3.5.5 Operations of Discrete Types

The basic operations of a discrete type include the operations involved in assignment.

Besides the basic operations, the operations of a discrete type include the predefined relational operators.

References: assignment 5.2, basic operation 3.3.3, discrete type 3.5, operation 3.3, relational operator 4.5 4.5.2, type 3.3

II-3.5.6 Real Types: Removed

II-3.5.7 Floating Point Types: Removed

II-3.5.8 Operations of Floating Point Types Types: Removed

II-3.5.9 Fixed Point Types: Removed

II-3.6 Array Types: Removed

II-3.7 Record Types: Removed

II-3.8 Access Types: Removed

II-3.9 Declarative Parts

A declarative part contains declarative items (possibly none).

```
declarative_part ::= {basic_declarative_item}          [Deno: 49,Syntax: 68,Static: 76,Dynamic: 81]

basic_declarative_item ::= object_declaration          [Deno: 49,Syntax: 68,Static: 76,Dynamic: 81]
```

The elaboration of a declarative part consists of the elaboration of the declarative items, if any, in the order in which they are given in the declarative part. After its elaboration, a declarative item is said to be elaborated. Prior to the completion of its elaboration (including before the elaboration), the declarative item is not yet elaborated.

For several forms of declarative item, the language rules (in particular scope and visibility rules) are such that it is either impossible or illegal to use an entity before the elaboration of the declarative item that declares this entity.

References: scope 8.2, visibility 8.3, object declaration 3.2.1,

Elaboration of declarations: 3.1, object declaration 3.2.1

Chapter II-4

NAMES AND EXPRESSIONS

The rules applicable to the different forms of name and expression, and to their evaluation, are given in this chapter.

II-4.1 Names

Names can denote declared entities.

```
   name ::= simple_name                                              [Syntax: 68]

   simple_name ::= identifier
```

A simple name for an entity is the identifier associated with the entity by its declaration.

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect.

Examples of simple names:

```
   LIMIT   --  the simple name of a constant             (see 3.2.1)
   COUNT   --  the simple name of a scalar variable      (see 3.2.1)
```

References: declaration 3.1, entity 3.1, evaluation 4.5, identifier 2.3, object 3.2.1

II-4.2 Literals: Removed

II-4.3 Aggregates: Removed

II-4.4 Expressions

An expression is a formula that defines the computation of a value.

```
   expression ::=  relation                [Deno: 49,Syntax: 69,Static: 78,Dynamic: 81]

   relation ::=                            [Deno: 49,Syntax: 69,Static: 78,Dynamic: 81]
      simple_expression  |
      simple_expression relational_operator simple_expression

   simple_expression ::= term
```

```
term ::= factor
factor ::= primary
primary ::= name
```

Each primary has a value and a type. The only names allowed as primaries are names denoting objects (the value of such a primary is the value of the object).

The type of an expression depends only on the type of its constituents and on the operators applied; for an overloaded constituent or operator, the determination of the constituent type, or the identification of the appropriate operator, is determinable by overload resolution (8.7). For each predefined operator, the operand and result types are given in section 4.5.

Examples of primaries:

```
SUM             --  variable
```

Examples of expressions:

```
VOLUME              -- primary
NATURAL_E < PI         -- expression
```

References: name 4.1, object 3.2, operator 4.5, overload resolution 8.7, relation 4.5.1, relational operator 4.5 4.5.2, type 3.3, variable 3.2.1

II-4.5 Operators and Expression Evaluation

The language defines the following class of operators:

```
relational_operator ::=  =   | /=  | <   | <= | > | >=          [Deno: 49,Syntax: 69,Static: 78,Dynamic: 81]
```

II-4.5.1 Logical Operatiors and Short Circuit Control Forms: Removed

II-4.5.2 Relational Operators

The equality and inequality operators are predefined for any type. The other relational operators are the ordering operators < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). The ordering operators are predefined for any scalar type. The operands of each predefined relational operator [must] have the same type. The result type is the predefined type BOOLEAN.

The relational operators have their conventional meaning: the result is equal to TRUE if the corresponding relation is satisfied; the result is FALSE otherwise. The inequality operator gives the complementary result to the equality operator: FALSE if equal, TRUE if not equal.

Operator	Operation	Operand type	Result type
= /=	equality and inequality	any type	BOOLEAN
< <= > >=	test for ordering	any scalar type	BOOLEAN

Equality for the discrete types is equality of the values.

References: boolean predefined type 3.5.3, operator 4.5, predefined operator 4.5, type 3.3

II-4.6 4.6 through 4.10 Removed

Chapter II-5

STATEMENTS

A statement defines an action to be performed; the process by which a statement achieves its action is called execution of the statement.

II-5.1 Simple and Compound Statements - Sequences of Statements

```
sequence_of_statements ::= statement {statement}     [Deno: 51,Syntax: 69,Static: 77,Dynamic: 81]

statement ::=  simple_statement                      [Deno: 49,Syntax: 69,Static: 77,Dynamic: 81]

simple_statement ::= null_statement                            [Syntax: 69,Static: 77,Dynamic: 81]
  | assignment_statement

null_statement ::= null;                             [Deno: 52,Syntax: 69,Static: 77,Dynamic: 81]
```

Execution of a null statement has no other effect than to pass to the next action.

The execution of a sequence of statements consists of the execution of the individual statements in succession until the sequence is completed.

References: assignment statement 5.2

II-5.2 Assignment Statement

An assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand side expression must be of the same type.

```
 assignment_statement ::=                            [Deno: 52,Syntax: 69,Static: 77,Dynamic: 81]
  variable_name := expression;
```

For the execution of an assignment statement, the variable name and the expression are first evaluated, in some order that is not defined by the language. Finally, the value of the expression becomes the new value of the variable.

Examples:

```
SHADE := BLUE;
FLAG := LOWER < HIGHER
```

References: evaluation 4.5, expression 4.4, name 4.1, type 3.3, variable 3.2.1

II-5.3 5.3 through 5.9: Removed

Chapter II-6
SUBPROGRAMS

A subprogram is a program unit whose execution is invoked as part of the execution of a program (see 10.1). There is one form of subprogram: procedures.

References: procedure 6.1

II-6.1 Subprogram Specifications

A subprogram specification defines the calling conventions of a procedure.

```
subprogram_specification ::=                    [Syntax: 69,Static: 76,Dynamic: 81]
    procedure identifier [formal_part]

formal_part ::=                                          [Syntax: 69,Static: 76]
  (parameter_specification {; parameter_specification})

parameter_specification ::=                                        [Syntax: 70]
  identifier_list : mode type_mark

mode ::= in out                                                    [Syntax: 70]
                                                                   [Static: 76]
```

The specification of a procedure specifies its identifier and its formal parameters (if any).

A parameter specification with several identifiers is equivalent to a sequence of single parameter specifications, as explained in section 3.2. Each single parameter specification declares a formal parameter.

The elaboration of a subprogram specification elaborates the corresponding formal part. The elaboration of a formal part has no other effect.

Examples of subprogram specifications:

```
procedure TRAVERSE_TREE
procedure INCREMENT(X : in out INTEGER)
```

References: elaboration 3.9, identifier 2.3, identifier list 3.2, elaboration has no other effect 3.1, procedure 6, type mark 3.3.2

II-6.2 Formal Parameter Modes: Removed

II-6.3 Subprogram Bodies

A subprogram body declares a procedure and specifies its execution.

```
  subprogram_body ::=                                [Deno: 49,Syntax: 70,Static: 76,Dynamic: 81]
      subprogram_specification is
         [declarative_part]
      begin
          sequence_of_statements
      end ;
```

[We have deleted the optional trailing simple name, due to AI-253.]

The elaboration of a subprogram body elaborates its subprogram specification.

The execution of a subprogram body is invoked as part of the execution of a program (see 10.1). For this execution the declarative part of the body is elaborated, and the sequence of statements of the body is then executed.

References: declaration 3.1, declarative part 3.9, elaboration 3.9, formal parameter 6.1, sequence of statements 5.1, subprogram 6, subprogram specification 6.1

II-6.4 6.4 through 6.7: Removed

Chapter II-7

PACKAGES: REMOVED

Chapter II-8

VISIBILITY RULES

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the program are described in this chapter. The formulation of these rules uses the notion of a declarative region.

References: declaration 3.1, declarative region 8.1, identifier 2.3, scope 8.2, visibility 8.3

II-8.1 Declarative Region

A declarative region is a portion of the program text. A single declarative region is formed by the text of each of the following:

- A subprogram body
- Package STANDARD

In the first of the above cases, the declarative region is said to be associated with the corresponding declaration. A declaration is said to occur immediately within a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

If any rule defines a portion of text as the text that extends from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region.

Notes:

As defined in section 3.1, the term declaration includes basic declarations, implicit declarations, and those declarations that are part of basic declarations, for example, parameter specifications. It follows from the definition of a declarative region that a parameter specification occurs immediately within the region associated with the enclosing subprogram body.

The package STANDARD forms a declarative region which encloses all library units (see sections 8.6 and 10.1).

Declarative regions can be nested within other declarative regions.

References: basic declaration 3.1, declaration 3.1, library unit 10.1, parameter specification 6.1, standard package 8.6, subprogram body 6.3

II-8.2 Scope of Declarations

For each form of declaration, the language rules define a certain portion of the program text called the scope of the declaration. The scope of a declaration is also called the scope of any entity declared by the declaration. Furthermore, if the declaration associates some notation with a declared entity, this portion of the text is also called the scope of this notation (either an identifier, an operator symbol, or the notation for a basic operation). Within the scope of an entity, and only there, there are places where it is legal to use the associated notation in order to refer to the declared entity. These places are defined by the rules of visibility and overloading.

The scope of a declaration that occurs immediately within a declarative region extends from the beginning of the declaration to the end of the declarative region; this part of the scope of a declaration is called the immediate scope.

Note:

The above scope rules apply to all forms of declaration defined by section 3.1; in particular, they apply also to implicit declarations.

References: basic operation 3.3.3, declaration 3.1, declarative region 8.1, extends 8.1, identifier 2.3, implicit declaration 3.1, occur immediately within 8.1, overloading 8.7, visibility 8.3

II-8.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules. The identifiers considered in this chapter include any identifier other than a reserved word. The places considered in this chapter are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this chapter are those for operations (including basic operations).

For each identifier and at each place in the text, the visibility rules determine a set of declarations (with this identifier) that define possible meanings of an occurrence of the identifier. A declaration is said to be visible at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. Two cases arise.

- The visibility rules determine at most one possible meaning. In such a case the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point.

- The visibility rules determine more than one possible meaning. In such a case the occurrence of the identifier is legal at this point if and only if exactly one visible declaration is acceptable for the overloading rules (see section 8.7 for overload resolution).

A declaration is only visible within a certain part of its scope; this part starts at the end of the declaration, except for a subprogram body, where it starts at the reserved word is appearing in the body. (This rule applies, in particular, for implicit declarations.)

A declaration is visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration, but excludes places where the declaration is hidden as explained below.

A declaration is said to be hidden within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a homograph of the other if both declarations have the same identifier.

Two declarations that occur immediately within the same declarative region must not be homographs.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the declared entity (if any) are also said to be visible from that point. Visibility is likewise defined for operator symbols. An operator is visible if and only if the corresponding operator declaration is visible. The notation associated with a basic operation is visible within the entire scope of this operation.

Note on immediate scope, hiding, and visibility:

The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier within its own declaration is illegal (except for a subprogram body). The identifier hides outer homographs within its immediate scope, that is, from the start of the declaration; on the other hand, the identifier is visible only after the end of the declaration. For this reason, all but the last of the following declarations are illegal:

```
K : INTEGER := K * K;        --  illegal
T : T;                       --  illegal
procedure R(R : INTEGER);    --  an inner declaration is legal
```

References: basic operation 3.3.3, declaration 3.1, declarative region 8.1, extends 8.1, identifier 2.3, immediate scope 8.2, lexical element 2.2, occur immediately within 8.1, reserved word 2.9, scope 8.2, subprogram 6, subprogram specification 6.1

II-8.4 Use Clauses: Removed

II-8.5 Renaming Declarations: Removed

II-8.6 The Package Standard

The predefined types BOOLEAN and INTEGER are implicitly declared in a declarative region called package STANDARD. The package STANDARD is described in Annex C.

The package STANDARD forms a declarative region which encloses every library unit and consequently the main program.

References: declaration 3.1, declarative region 8.1, implicit declaration 3.1, library unit 10.1, main program 10.1, occur immediately within 8.1, type 3.3

II-8.7 Overload Resolution

Overloading is defined for operators and also for the basic operation assignment. [We do not know what ‘‘assignment’’ and ‘‘assignment operations’’ are. Note inconsistency, within ARM, between 3.3.3(3) and here.]

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an operator [symbol] or [the notation for] some basic operation, whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence.

At such a place all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent.

When considering possible interpretations, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below.

(a) Any rule that requires a name or expression to have a certain type, or
to have the same type as another name or expression.

References: assignment 5.2, basic operation 3.3.3, class of type 3.3, declaration 3.1, expression 4.4, formal part 6.1, identifier 2.3, legal 1.6, name 4.1, operation 3.3.3, operator 4.5, statement 5, subprogram 6, visibility 8.3

Rules of the form (a): assignment 5.2

Chapter II-9

TASKS: REMOVED

Chapter II-10

PROGRAM STRUCTURE AND COMPILATION ISSUES

The overall structure of programs is described in this chapter. A program is a compilation unit submitted to a compiler in a compilation. A compilation unit specifies the compilation of a construct which is a subprogram body.

References: compilation 10.1, compilation unit 10.1, subprogram body 6.3

II-10.1 Compilation Units - Library Units

The text of a program is submitted to the compiler in a compilation. Each compilation is a compilation unit.

compilation ::= compilation_unit [Deno: 49,Syntax: 70,Static: 75,Dynamic: 81]

compilation_unit ::= library_unit [Deno: 49,Syntax: 70,Static: 75,Dynamic: 81]

library_unit ::= subprogram_body

For the visibility rules, each library unit acts as a declaration that occurs immediately within the package STANDARD.

A subprogram that is a library unit can be used as a main program in the usual sense. The means by which this execution is initiated are not prescribed by the language definition. An implementation may impose certain requirements on the parameters of a main program (these requirements must be stated in Appendix F). In any case, every implementation is required to allow, at least, main programs that are parameterless procedures, and every main program must be a subprogram that is a library unit.

Execution of a main program may be abandoned due to an implementation’s limitations.

The name of the main program may not be any of the following: INTEGER, BOOLEAN, FLOAT, CHARACTER, ASCII, NATURAL, POSITIVE, STRING, DURATION, CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, STORAGE_ERROR, TASKING_ERROR, TRUE, or FALSE. An implementation may further restrict the name of the program as recorded in Annex F.

References: allow 1.6, declaration 3.1, library unit 10.5, occur immediately within 8.1, parameter of a subprogram 6.2, procedure 6.1, program unit 6, standard package 8.6, subprogram 6, subprogram body 6.3, visibility 8.3

II-10.2 Subunits of Compilation Units: Removed

II-10.3 Order of Compilation: Removed

II-10.4 The Program Library: Removed

II-10.5 Elaboration of Library Units

Before the execution of a main program, its subprogram body is elaborated.

References: elaboration 3.1, main program 10.1, subprogram body 6.3

II-10.6 Program Optimization: Removed

Appendix A

Predefined Language Attributes: Removed

Appendix B
Predefined Language Pragmas: Removed

Appendix C
Predefined Language Environment

Package STANDARD contains implicit declarations of the predefined entities of n-AVA. These declared entities are:

- type BOOLEAN

- the boolean relational operators "=", "/=", "<",
  "<=", ">" and ">=" are defined.

- the basic operation inherent in assignment of booleans

- type INTEGER

- the integer relational operators "=", "/=", "<",
  "<=", ">" and ">=" are defined.

- the basic operation inherent in assignment of integers

References: assignment 5.2, basic operation 3.3.3, types 3.3

PART III: THE DENOTATIONAL SEMANTICS OF NANOAVA (VERSION 1)

Mark Saaltink

Chapter III-1
THE DENOTATIONAL DEFINITION

III-1.1 Notations

$\Pi(S)$	powerset of S
$f[x \leftarrow v]$	λ y . if $y=x$ then v else $f(y)$
$\langle x,y \rangle$	ordered pair
S^*	the set of sequences composed of elements of set S
Λ	empty sequence
$x; y$	prefix element x to sequence y
$x \bullet y$	append sequence y to sequence x

If i is defined as a variable ranging over set I, then i^* is implicitly defined as a variable ranging over I^*.

III-1.2 Abstract Syntax

$c \in$ Cmp	compilations	[ARM: 41,Syntax: 70,Static: 75,Dynamic: 81]
$p \in$ Sub	subprogram bodies	[ARM: 32,Syntax: 70,Static: 76,Dynamic: 81]
$bdi \in$ BDI	basic declarative items	[ARM: 23,Syntax: 68,Static: 76,Dynamic: 81]
$ps \in$ Ps	parameter specifications	[ARM: 31,Syntax: 70,Static: 76]
$s \in$ Stm	statements	[ARM: 29,Syntax: 69,Static: 77,Dynamic: 81]
$tm \in$ Tm	type marks	[ARM: 21,Syntax: 68,Static: 76,Dynamic: 81]
$e \in$ Exp	expressions	[ARM: 25,Syntax: 69,Static: 78,Dynamic: 81]
$O \in$ Opr	relational operators	[ARM: 26,Syntax: 69,Static: 78,Dynamic: 81]
$i \in$ Ide	identifiers	

c	::=	p
p	::=	**proc** $i\ ps^*\ bdi^*\ s^*$
ps	::=	**inout** $i^*\ tm$
bdi	::=	**var** $i^*\ tm\ e$ | **const** $i^*\ tm\ e$
s	::=	**null** | $i := e$
tm	::=	i
e	::=	i | $i\ O\ i$
O	::=	= | /= | < | <= | > | >=

III-1.3 Static Semantics

III-1.3.1 Domains

$di \in$ Din	declaration info
$\rho \in$ Env	static environments

di	::=	**type** $\mid$ **var**(tm) $\mid$ **const**(tm) $\mid$ **proc**
Env	=	Π(Ide) x (Ide $\Rightarrow$ Π(Din))

The static environment used in the analysis of some part of a program is intended to represent the declarations visible in that part of the program. In this formalism, we record some information derived from the visible declarations rather than the declarations themselves. We allow for a set of declarations that might define a meaning of an identifier (although this set can contain at most one member in n-AVA). It is also necessary to keep track of the names declared immediately within the current declarative region in order to be able to check compliance with rule [ARM 8.3(17)]. Therefore, the environment has two parts: a set of locally declared names, and a mapping from identifiers to the set of (declaration info associated with) visible declarations of this identifier.

The environment is not used to associate meanings with ‘‘notations associated with basic operations’’; it should be in order to more closely correspond to the rules as they are expressed in the ARM.

Five basic functions act on environments:

locals:	Env $\Rightarrow$ Π(Ide)
lookup:	Ide x Env $\Rightarrow$ Π(Din)
start_declarative_region:	Env $\Rightarrow$ Env
hide_homographs:	Ide x Env $\Rightarrow$ Env
add_decl:	Ide x Din x Env $\Rightarrow$ Env

Functions ‘‘locals’’ and ‘‘lookup’’ merely extract the appropriate component of the environment. Function ‘‘start_declarative_region’’ modifies the environment when a new declarative region is entered. Function ‘‘hide_homographs’’ removes homographs of a particular identifier (when the scope of a new declaration is entered), and function ‘‘add_decl’’ adds the declaration info associated with a declaration (when the declaration becomes visible).

locals($\langle l,m\rangle$) = l
lookup(i, $\langle l,m\rangle$) = $m(i)$
start_declarative_region($\langle l,m\rangle$)= $\langle \phi,m\rangle$
hide_homographs(i, $\langle l,m\rangle$) = $\langle l, m\,[i \leftarrow \phi]\rangle$
add_decl(i, di, $\langle l,m\rangle$) = $\langle l \cup \{ i \} , m\,[i \leftarrow m(i) \cup \{ di \}]\rangle$

Relating the formal static semantics with the narrative rules of scoping and visibility is not trivial. Two ideas are central: (1) the basic functions on the environment modify it in appropriate ways (reflecting the correct locals and visible declarations after each event mentioned informally above); and (2) these functions are applied appropriately in the static semantics (so that an appropriate environment in used in the analysis of each part of a program).

We also use two derived syntactic domains:

$nps \in$ nPs	normalized parameter specifications
$nbdi \in$ nBDI	normalized basic declarative items

nps	::=	**inout** *i tm*
nbdi	::=	**var** *i tm e* | **const** *i tm e*

Various normalization functions act on these domains:

N_{ps^*} :	$\mathrm{Ps}^* \Rightarrow \mathrm{nPs}^*$
N_{ps} :	$\mathrm{Ps} \Rightarrow \mathrm{nPs}^*$
N_{bdi^*} :	$\mathrm{BDI}^* \Rightarrow \mathrm{nBDI}^*$
N_{bdi} :	$\mathrm{BDI} \Rightarrow \mathrm{nBDI}^*$

These new domains and normalization functions are used to reflect the rules [ARM 3.2(10)] on multiple object declarations and [ARM 6.1(4)] on multiple parameter specifications.

$N_{ps^*}(\Lambda) = \Lambda$
$N_{ps^*}(ps;\, ps^*) = N_{ps}(ps) \bullet N_{ps^*}(ps^*)$
$N_{ps}(\textbf{inout}\ \Lambda\ tm) = \Lambda$
$N_{ps}(\textbf{inout}\ (i;i^*)\ tm) = (\{\textbf{inout}\ i\ tm\});\ N_{ps}(\textbf{inout}\ i^*\ tm)$

$N_{bdi^*}(\Lambda) = \Lambda$
$N_{bdi^*}(bdi;\, bdi^*) = N_{bdi}(bdi) \bullet N_{bdi^*}(bdi^*)$

$N_{bdi}(\textbf{var}\ \Lambda\ tm\ e) = \Lambda$
$N_{bdi}(\textbf{var}\ (i;i^*)\ tm\ e) = (\textbf{var}\ i\ tm\ e);\ N_{bdi}(\textbf{var}\ i^*\ tm\ e)$
$N_{bdi}(\textbf{const}\ \Lambda\ tm\ e) = \Lambda$
$N_{bdi}(\textbf{const}\ (i;i^*)\ tm\ e) = (\textbf{const}\ i\ tm\ e);\ N_{bdi}(\textbf{const}\ i^*\ tm\ e)$

III-1.3.2 Static semantics functions

The following functions are used in well-formedness testing:

W_c :	$\mathrm{Cmp} \Rightarrow \mathrm{Bool}$	[ARM: 41,Syntax: 70,Static: 75,Dynamic: 81]
W_{sub} :	$\mathrm{Sub} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 32,Syntax: 70,Static: 76,Dynamic: 81]
W_{ps^*} :	$\mathrm{Ps}^* \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 31,Syntax: 70,Static: 76]
W_{nps^*} :	$\mathrm{nPs}^* \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	
W_{nps} :	$\mathrm{nPs} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	
W_{bdi^*} :	$\mathrm{BDI}^* \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 23,Syntax: 68,Static: 76,Dynamic: 81]
W_{nbdi^*} :	$\mathrm{nBDI}^* \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	
W_{nbdi} :	$\mathrm{nBDI} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	
W_{s^*} :	$\mathrm{Stm}^* \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 29,Syntax: 69,Static: 77,Dynamic: 81]
W_s :	$\mathrm{Stm} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	
W_{tm} :	$\mathrm{Tm} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 21,Syntax: 68,Static: 76,Dynamic: 81]
W_e :	$\mathrm{Exp} \times \mathrm{Tm} \times \mathrm{Env} \Rightarrow \mathrm{Bool}$	[ARM: 25,Syntax: 69,Static: 78,Dynamic: 81]

The only unusual functions here are W_c and W_e. $W_c(c)$ is true iff c is a well-formed compilation unit. $W_e(e, t, \rho)$ is true iff e is a well-formed expression of type t with respect to environment ρ. Each of the other functions is true iff its first argument is well-formed with respect to the second argument (the environment).

The following functions are used in expressing modifications of the environment:

X_{sub} :	Sub x Env ⇒ Env
X_{ps^*} :	Ps^* x Env ⇒ Env
X_{nps^*} :	nPs^* x Env ⇒ Env
X_{nps} :	nPs x Env ⇒ Env
X_{bdi^*} :	BDI^* x Env ⇒ Env
X_{nbdi^*} :	nBDI^* x Env ⇒ Env
X_{nbdi} :	nBDI x Env ⇒ Env

III-1.3.3 Static Semantic Definitions

$W_c(c) = W_{sub}(c, \langle l,m \rangle)$
where l = { INTEGER, BOOLEAN, FLOAT, ... }
and $m = (\lambda\, i \in \text{Ide} \;.\; \phi)$ [BOOLEAN ← { **type** } , INTEGER ← { **type** }]

$W_{sub}(\textbf{proc}\ i\ ps^*\ bdi^*\ s^*, \rho)$ =
let ρ’ = hide_homographs(i, start_declarative_region(ρ))
and ρ’’ = $X_{sub}(\textbf{proc}\ i\ ps^*\ bdi^*\ s^*, \rho')$ in
$i \notin$ locals(ρ)
$\wedge\ W_{ps^*}(ps^*, \rho')$
$\wedge\ W_{bdi^*}(\text{bdi}^*, \text{X}_{\text{ps}^*}(\text{ps}^*, \rho''))$
$\wedge\ W_{s^*}(\text{s}^*, \text{X}_{\text{bdi}^*}(\text{bdi}^*, \text{X}_{\text{ps}^*}(\text{ps}^*, \rho'')))$

$W_{ps^*}(\text{ps}^*, \rho) = W_{nps^*}(N_{ps^*}(\text{ps}^*), \rho)$

$W_{nps^*}(\Lambda, \rho) = \textit{true}$
$W_{nps^*}(nps;\ nps^*, \rho) = W_{nps}(nps, \rho) \wedge W_{nps^*}(nps^*, X_{nps}(nps, \rho))$

$W_{nps}(\textbf{inout}\ i\ tm, \rho) = i \notin \text{locals}(\rho) \wedge W_{tm}(tm, \text{hide\_homographs}(i, \rho))$

$W_{bdi^*}(\text{bdi}^*, \rho) = W_{nbdi^*}(N_{bdi^*}(\text{bdi}^*), \rho)$

$W_{nbdi^*}(\Lambda, \rho) = \text{true}$

$W_{nbdi^*}(\text{nbdi};\ \text{nbdi}^*, \rho) = W_{bdi}(\text{nbdi}, \rho) \wedge W_{nbdi^*}(nbdi^*, X_{nbdi}(nbdi, \rho))$
$W_{nbdi}(\textbf{var}\ i\ tm\ e, \rho)$ =
$W_{nbdi}(\textbf{const}\ i\ tm\ e, \rho)$ =
$i \notin$ locals(ρ)
$\wedge\ W_{tm}(\text{tm}, \text{hide\_homographs}(i, \rho))$
$\wedge\ W_e(e, tm, \text{hide\_homographs}(i, \rho))$

$W_{s^*}(\Lambda, \rho) = \text{true}$
$W_{s^*}(\text{s};\ \text{s}^*, \rho) = W_s(\text{s}, \rho) \wedge W_{s^*}(\text{s}^*, \rho)$

$W_s(\textbf{null}, \rho) = \text{true}$ [ARM: 29,Syntax: 69,Static: 77,Dynamic: 81]
$W_s(i := e, \rho) = \exists\, t \in \text{Tm} \;.\; \text{lookup}(i, \rho) = \{ \textbf{var}(t) \} \wedge W_e(e, t, \rho)$
[ARM: 29,Syntax: 69,Static: 77,Dynamic: 81]

$W_{tm}(i, \rho) = (\text{lookup}(i, \rho) = \{ \textbf{type} \})$

$W_e(i, tm, \rho) = \text{lookup}(i, \rho) \in \{ \{\textbf{var}(tm)\}, \{\textbf{const}(tm)\} \}$
$W_e(i\ O\ i', tm, \rho) = (tm = \text{BOOLEAN}) \wedge \exists\, tm' \;.\; W_e(i, tm', \rho) \wedge W_e(i', tm', \rho)$

$X_{sub}(\textbf{proc}\ i\ ps^{*}\ bdi^{*}\ s^{*}, \rho) = \text{add\_decl}(i, \textbf{proc}, \rho)$

$X_{ps^*}(ps^{*}, \rho) = X_{nps^*}(N_{ps^*}(ps^{*}), \rho)$

$X_{nps^*}(\Lambda, \rho) = \rho$
$X_{nps^*}(nps;\ nps^{*}, \rho) = X_{nps^*}(nsp^{*}, X_{nps}(nps, \rho))$

$X_{nps}(\textbf{inout}\ i\ tm, \rho) = \text{add\_decl}(i, \textbf{var}(tm), \text{hide\_homographs}(i, \rho))$

$X_{bdi^*}(bdi^{*}, \rho) = X_{nbdi^*}(N_{bdi^*}(bdi^{*}), \rho)$

$X_{nbdi^*}(\Lambda, \rho) = \rho$
$X_{nbdi^*}(nbdi;\ nbdi^{*}, \rho) = X_{nbdi^*}(nbdi^{*}, X_{nbdi}(nbdi, \rho))$

$X_{nbdi}(\textbf{const}\ i\ tm\ e, \rho) = \text{add\_decl}(i, \textbf{const}(tm), \text{hide\_homographs}(i, \rho))$
$X_{nbdi}(\textbf{var}\ i\ tm\ e, \rho) = \text{add\_decl}(i, \textbf{var}(tm), \text{hide\_homographs}(i, \rho))$

III-1.4 Dynamic Semantics

The dynamic semantics of n-AVA are very simple. The significant domain is the *state*, mapping identifiers to their values. The state collects together the current values of all objects, and is changed to reflect changes in the values of any objects. For nanoAVA, we can take objects to be the same as identifiers.

The nanoAVA reference manual says little about how a compilation unit (subprogram) is invoked. The formal definition captures the effects described in [ARM 6.3]. However, we have no notion of ‘‘calling’’ the main procedure; rather, we just execute the subprogram body. The initial state for this execution is presumed to be established in some implementation-dependent way (and so is not specified by the formal definition).

III-1.4.1 Domains

$v \in Val$	values
$\sigma \in \Sigma$	states

Val	$= \text{Bool} \cup \text{Int}$
Σ	$= \text{Ide} \Rightarrow \text{Val}$

III-1.4.2 Dynamic Functions (signatures)

The following functions express the effect of elaborating, evaluating, or executing the various constructs:

E_c :	$\text{Cmp} \times \Sigma \Rightarrow \Sigma$
E_{bdi^*} :	$\text{BDI}^{*} \times \Sigma \Rightarrow \Sigma$
E_{nbdi^*} :	$\text{nBDI}^{*} \times \Sigma \Rightarrow \Sigma$
E_{bdi} :	$\text{nBDI} \times \Sigma \Rightarrow \Sigma$
E_{s^*} :	$\text{Stm}^{*} \times \Sigma \Rightarrow \Sigma$
E_s :	$\text{Stm} \times \Sigma \Rightarrow \Sigma$
E_e :	$\text{Exp} \times \Sigma \Rightarrow \text{Val}$
E_o :	$\text{Opr} \times \text{Val} \times \text{Val} \Rightarrow \text{Val}$

III-1.4.3 Dynamic Semantics Definitions

$E_c($**proc** $i\ ps^*\ bdi^*\ s^*, \sigma) = E_{s^*}(\mathrm{s}^*, E_{bdi^*}(bdi^*, \sigma))$

$E_{bdi^*}(\mathrm{bdi}^*, \sigma) = E_{nbdi^*}(N_{bdi^*}(bdi^*), \sigma)$

$E_{nbdi^*}(\Lambda, \sigma) = \sigma$
$E_{nbdi^*}(nbdi;\ nbdi^*, \sigma) = E_{nbdi^*}(nbdi^*, E_{nbdi}(nbdi, \sigma))$

$E_{nbdi}($**var** $i\ tm\ e, \sigma) =$
$E_{nbdi}($**const** $i\ tm\ e, \sigma) = \sigma\ [\ i \leftarrow E_e(e, \sigma)\ ]$

$E_{s^*}(\Lambda, \sigma) = \sigma$
$E_{s^*}(s;\ s^*, \sigma) = E_{s^*}(s^*, E_s(s, \sigma))$

$E_s($**null**$, \sigma) = \sigma$
$E_s(i := e, \sigma) = \sigma\ [\ i \leftarrow E_e(e, \sigma)\ ]$

$E_e(i, tm, \sigma) = \sigma(i)$
$E_e(i\ O\ i', tm, \sigma) = E_o(O, E_e(i, \sigma), E_e(i', \sigma))$

$E_o(\ =, v, v')$ = if $v = v'$ then true else false
$E_o(\ /=, v, v')$ = if $v \neq v'$ then true else false
$E_o(\ <, v, v')$ = if $v < v'$ in Int $\vee$ (v = false $\wedge$ v' = true) then true else false
$E_o(\ <=, v, v')$ = if $v <= v'$ in Int $\vee$ $(v = \mathit{false} \vee v' = \mathit{true})$ then true else false
$E_o(\ >, v, v')$ = if $v > v'$ in Int $\vee$ $(v = \mathit{true} \wedge v' = \mathit{false})$ then true else false
$E_o(\ >=, v, v')$ = if $v >= v'$ in Int $\vee$ $(v = \mathit{true} \vee v' = \mathit{false})$ then true else false

III-1.5 Inadequacies

Several aspects of this definition are disturbing, or will not work in a larger subset of Ada:

- There is no dynamic environment. As soon as we have subprograms and calls, we will need one. To conform to the form of the ARM, we should perhaps use a dynamic environment to record which declarations have been elaborated yet (although this doesn't appear to matter in n-AVA).
- Should there be a domain of ‘‘interpretations’’? (This would correspond to Mike's second prefix form.) Interpretations would let us reflect the overloading rules, and would allow us to pass extra information to the dynamic semantics. (For example, we could disambiguate the relational operators.)
- Some of the environment manipulations (such as hiding homographs) appear in both the *W* functions and the *X* functions. It would be nice to avoid this duplication.
- In the ARM, parameter specifications and basic declarative items are lumped under declarations, with general rules applying to both. Should that be reflected in the formalism?
- There is a lack of parallelism in the definitions of X_{sub} on one hand, and X_{nps} and X_{nbdi} on the other. (This may relate to the preceding item.)
- Should the static environment contain info for ‘‘notations associated with basic operations’’ [ARM 8.13 (18)]?

Chapter III-2
THE BOYER MOORE DEFINITION

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10 -*-

(boot-strap)

;;; some basic functions
;;;

(defn append (x y)
  (if (listp x)
      (cons (car x) (append (cdr x) y))
      y))

(defn a-lookup (key alist not-found)
  (if (listp alist)
      (if (and (listp (car alist))
               (equal key (caar alist)))
          (cdar alist)
          (a-lookup key (cdr alist) not-found))
      not-found))

;;; finite functions
;;;
;;; we have functions that are constant except on a finite subdomain
;;; represented by (constant-value . exception-alist)

(defn constant-function (val)
  (cons val nil))

(defn apply-function (fn x)
  (a-lookup x (cdr fn) (car fn)))

(defn update-function (fn x val)
  (cons (car fn)
        (cons (cons x val) (cdr fn))))

(prove-lemma apply-constant-function (rewrite)
  (equal (apply-function (constant-function c) x)
         c))

(prove-lemma apply-update-function (rewrite)
  (equal (apply-function (update-function fn x v) y)
         (if (equal x y)
             v
             (apply-function fn y))))

(disable constant-function)
(disable apply-function)
(disable update-function)

;;; sets
```

```
;;;

(defn set-p (x)
  (if (listp x)
      (and (not (member (car x) (cdr x)))
           (set-p (cdr x)))
      (equal x nil)))

(defn set-size (x)
  (if (listp x)
      (add1 (set-size (cdr x)))
      0))

(defn null-set () nil)

(defn unit-set (x) (cons x nil))

(defn unit-set-p (x)
  (and (listp x)
       (equal (cdr x) nil)))

(defn unit-set-member (x)
  (car x))

(defn set-intersection (x y)
  (if (listp x)
      (if (member (car x) y)
          (cons (car x) (set-intersection (cdr x) y))
          (set-intersection (cdr x) y))
      nil))

(defn set-union (x y)
  (if (listp x)
      (if (member (car x) y)
          (set-union (cdr x) y)
          (cons (car x) (set-union (cdr x) y)))
      y))

(prove-lemma member-null-set (rewrite)
  (equal (member x (null-set))
         (false)))

(prove-lemma member-unit-set (rewrite)
  (equal (member x (unit-set y))
         (equal x y)))

(prove-lemma member-set-intersection (rewrite)
  (implies (and (set-p x) (set-p y))
           (equal (member a (set-intersection x y))
                  (and (member a x)
                       (member a y)))))

(prove-lemma member-set-union (rewrite)
  (implies (and (set-p x) (set-p y))
           (equal (member a (set-union x y))
                  (or (member a x)
                      (member a y)))))

(prove-lemma unit-set-p-unit-set (rewrite)
  (unit-set-p (unit-set x)))

(prove-lemma unit-set-p-characterization ()
  (equal (unit-set-p x)
         (equal x (unit-set (unit-set-member x)))))
```

III-2.1 Abstract syntax

```
;;; macros for declaring record-like structures
;;;
;;; (defrecord foo (bar baz ...)) expands to
;;;    (defn mk-foo (bar baz ...) (list 'foo bar baz ...))
;;;    (defn foo-p (x) (and (listp x) (equal (car x) 'foo)))
;;;    (defn foo-bar (x) (car (cdr x)))
;;;    (defn foo-baz (x) (car (cdr (cdr x))))
;;;    ...

(defmacro defrecord (name  fields)
  `(progn
     (defn ,(intern (string-upcase (string-append "mk-" (string name)))) ,fields
       (list ',name ,@fields))
     (defn ,(intern (string-upcase (string-append (string name) "-p"))) (x)
       (and (listp x) (equal (car x) ',name)))
     ,@(declare-fields (string name) fields '(cdr x))))

(defun declare-fields (name fields acc)
  (if (null fields)
      nil
      (cons `(defn ,(intern (string-upcase (string-append name "-" (string (car fields)))))
                   (x) (car ,acc))
            (declare-fields name (cdr fields) (list 'cdr acc)))))

;;; abstract syntax

(defrecord opr-eq ())
(defrecord opr-ne ())
(defrecord opr-lt ())
(defrecord opr-le ())
(defrecord opr-gt ())
(defrecord opr-ge ())

(defrecord exp-ide (ide))
(defrecord exp-rel (lhs opr rhs))

(defrecord stm-null ())
(defrecord stm-asg (var exp))

(defrecord ps (ide-list tm))

(defrecord bdi-const (ide-list tm exp))
(defrecord bdi-var (ide-list tm exp))

(defrecord sub (ide ps-list bdi-list stm-list))
```

III-2.2 Normalization

```
(defrecord nps (ide tm))

(defrecord nbdi-const (ide tm exp))
(defrecord nbdi-var (ide tm exp))

(defn n-ps-sub (i-list tm)
  (if (listp i-list)
      (cons (mk-nps (car i-list) tm)
            (n-ps-sub (cdr i-list) tm))
      nil))

(defn n-ps (x) (n-ps-sub (ps-ide-list x) (ps-tm x)))

(defn n-ps-list (x)
  (if (listp x)
      (append (n-ps (car x)) (n-ps-list (cdr x)))
      nil))

(prove-lemma n-ps-alt-def ()  ; this shows equivalence to the defn in the DS
  (equal (n-ps ps)
         (if (listp (ps-ide-list ps))
             (cons (mk-nps (car (ps-ide-list ps)) (ps-tm ps))
                   (n-ps (mk-ps (cdr (ps-ide-list ps)) (ps-tm ps))))
             nil)))

(defn n-bdi-const (i-list tm e)
  (if (listp i-list)
      (cons (mk-nbdi-const (car i-list) tm e)
            (n-bdi-const (cdr i-list) tm e))
      nil))

(defn n-bdi-var (i-list tm e)
  (if (listp i-list)
      (cons (mk-nbdi-var (car i-list) tm e)
            (n-bdi-var (cdr i-list) tm e))
      nil))

(defn n-bdi (x)
  (if (bdi-const-p x)
      (n-bdi-const (bdi-const-ide-list x) (bdi-const-tm x) (bdi-const-exp x))
      (n-bdi-var (bdi-var-ide-list x) (bdi-var-tm x) (bdi-var-exp x))))


(defn n-bdi-list (x)
  (if (listp x)
      (append (n-bdi (car x)) (n-bdi-list (cdr x)))
      nil))
```

III-2.3 Static Domains

```
;;; declaration info is represented using records

(defrecord di-type ())
(defrecord di-proc ())
(defrecord di-const (tm))
(defrecord di-var (tm))

;;; envs are pairs (set of local, mapping from ide to set of decl info)
;;; using the representations defined above

(defn env-locals (env) (car env))

(defn env-lookup (i env) (apply-function (cdr env) i))

(defn start-dr (env) (cons (null-set)
                           (cdr env)))

(defn hide-homog (i env) (cons (car env)
                               (update-function (cdr env) i (null-set))))

(defn add-decl (i di env) (cons (set-union (unit-set i) (env-locals env))
                                (update-function (cdr env)
                                                 i
                                                 (set-union (unit-set di)
                                                            (env-lookup i env)))))
;; basic properties

(prove-lemma env-locals-start-dr ()
  (equal (env-locals (start-dr env))
         (null-set)))

(prove-lemma env-lookup-start-dr ()
  (equal (env-lookup i (start-dr env))
         (env-lookup i env)))

(prove-lemma env-locals-hide-homog ()
  (equal (env-locals (hide-homog i env))
         (env-locals env)))

(prove-lemma env-lookup-hide-homog ()
  (equal (env-lookup i (hide-homog j env))
         (if (equal i j)
             (null-set)
             (env-lookup i env))))

(prove-lemma env-locals-add-decl ()
  (equal (env-locals (add-decl i di env))
         (set-union (unit-set i) (env-locals env))))

(prove-lemma env-lookup-add-decl ()
  (equal (env-lookup i (add-decl j di env))
         (if (equal i j)
             (set-union (unit-set di) (env-lookup i env))
             (env-lookup i env))))
```

III-2.4 Well-formedness predicates

```
;;; X (extension) function

(defn x-nbdi (nbdi env)
  (if (nbdi-const-p nbdi)
      (add-decl (nbdi-const-ide nbdi)
                (mk-di-const (nbdi-const-tm nbdi))
                (hide-homog (nbdi-const-ide nbdi) env))
      (add-decl (nbdi-var-ide nbdi)
                (mk-di-var (nbdi-var-tm nbdi))
                (hide-homog (nbdi-var-ide nbdi) env))))

(defn x-nbdi-list (nbdi-list env)
  (if (listp nbdi-list)
      (x-nbdi-list (cdr nbdi-list) (x-nbdi (car nbdi-list) env))
      env))

(defn x-bdi-list (bdi-list env)
  (x-nbdi-list (n-bdi-list bdi-list) env))

(defn x-nps (nps env)
  (add-decl (nps-ide nps)
            (mk-di-var (nps-tm nps))
            (hide-homog (nps-ide nps) env)))

(defn x-nps-list (nps-list env)
  (if (listp nps-list)
      (x-nps-list (cdr nps-list) (x-nps (car nps-list) env))
      env))

(defn x-ps-list (ps-list env)
  (x-nps-list (n-ps-list ps-list) env))

(defn x-sub (sub env)
  (add-decl (sub-ide sub)
            (mk-di-proc)
            env))
```

```
;;; W (well-formedness) functions
;;;
;;; W-exp has been modified; rather than taking a type-mark and returning a boolean,
;;; it returns either nil or (cons 'type tm)
;;; so that We(e, tm, env) is equivalent to  (equal (w-exp e env) (cons 'type tm))

(defn w-exp (e env)
  (if (exp-rel-p e)
      (if (and (w-exp (exp-rel-lhs e) env)
               (equal (w-exp (exp-rel-lhs e) env)
                      (w-exp (exp-rel-rhs e) env)))
          (cons 'type 'boolean)
          nil)
      (if (and (unit-set-p (env-lookup (exp-ide-ide e) env))
               (or (di-const-p (unit-set-member (env-lookup (exp-ide-ide e) env)))
                   (di-var-p (unit-set-member (env-lookup (exp-ide-ide e) env)))))
          (cons 'type
                (if (di-const-p (unit-set-member (env-lookup (exp-ide-ide e) env)))
                    (di-const-tm (unit-set-member (env-lookup (exp-ide-ide e) env)))
                    (di-var-tm (unit-set-member (env-lookup (exp-ide-ide e) env)))))
          nil)))

(defn w-tm (tm env)
  (equal (env-lookup tm env)
         (unit-set (mk-di-type))))

(defn w-s (s env)
  (or (stm-null-p s)
      (and (stm-asg-p s)
           (unit-set-p (env-lookup (stm-asg-var s) env))
           (di-var-p (unit-set-member (env-lookup (stm-asg-var s) env)))
           (equal (w-exp (stm-asg-exp s) env)
                  (cons 'type
                        (di-var-tm (unit-set-member (env-lookup (stm-asg-var s) env))))))))


(defn w-s-list (s-list env)
  (if (listp s-list)
      (and (w-s (car s-list) env)
           (w-s-list (cdr s-list) env))
      (equal s-list nil)))

(defn w-nbdi (x env)
  (if (nbdi-const-p x)
      (and (not (member (nbdi-const-ide x) (env-locals env)))
           (w-tm (nbdi-const-tm x) (hide-homog (nbdi-const-ide x) env))
           (equal (w-exp (nbdi-const-exp x) (hide-homog (nbdi-const-ide x) env))
                  (cons 'type (nbdi-const-tm x))))
      (and (not (member (nbdi-var-ide x) (env-locals env)))
           (w-tm (nbdi-var-tm x) (hide-homog (nbdi-var-ide x) env))
           (equal (w-exp (nbdi-var-exp x) (hide-homog (nbdi-var-ide x) env))
                  (cons 'type (nbdi-var-tm x))))))


(defn w-nbdi-list (nbdi-list env)
  (if (listp nbdi-list)
      (and (w-nbdi (car nbdi-list) env)
           (w-nbdi-list (cdr nbdi-list) (x-nbdi (car nbdi-list) env)))
      (equal nbdi-list nil)))

(defn w-bdi-list (bdi-list env) (w-nbdi-list (n-bdi-list bdi-list) env))

(defn w-nps (nps env)
  (and (not (member (nps-ide nps) (env-locals env)))
       (w-tm (nps-tm nps) (hide-homog (nps-ide nps) env))))

(defn w-nps-list (nps-list env)
  (if (listp nps-list)
      (and (w-nps (car nps-list) env)
           (w-nps-list (cdr nps-list) (x-nps (car nps-list) env)))
      (equal nps-list nil)))

(defn w-ps-list (ps-list env) (w-nps-list (n-ps-list ps-list) env))

(defn w-sub (x env)
  (and (not (member (sub-ide x) (env-locals env)))
       (w-ps-list (sub-ps-list x)
                  (hide-homog (sub-ide x) (start-dr env)))
       (w-bdi-list (sub-bdi-list x)
                   (x-ps-list (sub-ps-list x)
                              (hide-homog (sub-ide x) (start-dr env))))
       (w-s-list (sub-stm-list x)
                 (x-bdi-list (sub-bdi-list x)
                             (x-ps-list (sub-ps-list x)
                                        (hide-homog (sub-ide x) (start-dr env)))))))

(defn w-c (c)
  (w-sub c
         (add-decl 'boolean (mk-di-type)
                   (add-decl 'integer (mk-di-type)
                             (cons '(float character ascii natural positive string duration
                                     constraint-error numeric-error program-error
                                     storage-error tasking-error true false)
                                   (constant-function (null-set)))))))
```

III-2.5 Test Cases for the Static Semantics

```
;;; test cases for the static semantics

(prove '(w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp))))))

(prove '(w-c (mk-sub 'swap-2
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(boolean) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'boolean))))))

(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(boolean)
                                         'boolean       ; should fail
                                         (mk-exp-rel (mk-exp-ide'x)
                                                     (mk-opr-ne)
                                                     (mk-exp-ide 'y))))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'y)))))))


(prove '(not (w-c (mk-sub 'integer                      ; should fail
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp)))))))

(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'float))       ; should fail
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp)))))))
(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'temp)))  ; should fail
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp)))))))

(prove '(w-c (mk-sub 'swap
                     (list (mk-ps '(swap y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'swap)))
                     (list (mk-stm-asg 'swap (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp))))))

(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'temp (mk-exp-ide 'y)))))))      ; should fail


(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x x) 'integer))     ; should fail
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'x))
                           (mk-stm-asg 'x (mk-exp-ide 'temp)))))))

(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(y) 'integer (mk-exp-ide 'x))) ; should fail
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'y)))))))

(prove '(not (w-c (mk-sub 'swap
                     (list (mk-ps '(x y) 'integer))
                     (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x))
                           (mk-bdi-var '(temp) 'integer (mk-exp-ide 'y)))       ; should fail
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'y (mk-exp-ide 'temp)))))))

(prove '(w-c (mk-sub 'compare
                     (list (mk-ps '(x y) 'integer)
                           (mk-ps '(b) 'boolean))
                     (list (mk-bdi-const '(temp)
                                         'boolean
                                         (mk-exp-rel (mk-exp-ide 'x)
                                                     (mk-opr-le)
                                                     (mk-exp-ide 'y)))
                           (mk-bdi-var '(xx yy) 'boolean (mk-exp-ide 'temp)))
                     (list (mk-stm-asg 'x (mk-exp-ide 'y))
                           (mk-stm-asg 'b (mk-exp-ide 'xx))
                           (mk-stm-null)
                           (mk-stm-asg 'b (mk-exp-ide 'yy))))))
```

III-2.6 Dynamic Semantics

```
;;;
;;; dynamic semantics
;;;

;;; states are represented by finite functions

(defn e-opr (opr v1 v2)
  (if (opr-eq-p opr) (if (equal v1 v2) 'true 'false)
  (if (opr-ne-p opr) (if (not (equal v1 v2)) 'true 'false)
  (if (opr-lt-p opr) (if (or (and (equal v1 'false) (equal v2 'true))
                             (lessp v1 v2))
                         'true
                         'false)
  (if (opr-le-p opr) (if (or (or (equal v1 'false) (equal v2 'true))
                             (equal v1 v2)
                             (lessp v1 v2))
                         'true
                         'false)
  (if (opr-gt-p opr) (if (or (and (equal v1 'true) (equal v2 'false))
                             (lessp v2 v1))
                         'true
                         'false)
  (if (opr-ge-p opr) (if (or (or (equal v1 'true) (equal v2 'false))
                             (equal v1 v2)
                             (lessp v1 v2))
                         'true
                         'false)
  0)))))))

(defn e-e (x store)
  (if (exp-rel-p x)
      (e-opr (exp-rel-opr x)
             (e-e (exp-rel-lhs x) store)
             (e-e (exp-rel-rhs x) store))
      (apply-function store (exp-ide-ide x))))

(defn e-s (x store)
  (if (stm-null-p x)
      store
      (update-function store
                       (stm-asg-var x)
                       (e-e (stm-asg-exp x) store))))

(defn e-s-list (x store)
  (if (listp x)
      (e-s-list (cdr x)
                (e-s (car x) store))
      store))

(defn e-nbdi (x store)
  (if (nbdi-const-p x)
      (update-function store
                       (nbdi-const-ide x)
                       (e-e (nbdi-const-exp x) store))
      (update-function store
                       (nbdi-var-ide x)
                       (e-e (nbdi-var-exp x) store))))

(defn e-nbdi-list (x store)
  (if (listp x)
      (e-nbdi-list (cdr x)
                   (e-nbdi (car x) store))
      store))

(defn e-bdi-list (x store)
  (e-nbdi-list (n-bdi-list x) store))

(defn e-c (x store)
  (e-s-list (sub-stm-list x)
            (e-bdi-list (sub-bdi-list x) store)))
```

III-2.7 Test Cases for the Dynamic Semantics

```
;;; test cases for the dynamic semantics

(prove '(equal
          (apply-function
            (e-c (mk-sub 'swap
                         (list (mk-ps '(x y) 'integer))
                         (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                         (list (mk-stm-asg 'x (mk-exp-ide 'y))
                               (mk-stm-asg 'y (mk-exp-ide 'temp))))
                 store)
            'x)
          (apply-function store 'y)))

(prove '(equal
          (apply-function
            (e-c (mk-sub 'swap
                         (list (mk-ps '(x y) 'integer))
                         (list (mk-bdi-const '(temp) 'integer (mk-exp-ide 'x)))
                         (list (mk-stm-asg 'x (mk-exp-ide 'y))
                               (mk-stm-asg 'y (mk-exp-ide 'temp))))
                 store)
            'y)
          (apply-function store 'x)))

(prove-lemma stm-cant-clobber-const (rewrite)
  (implies (and (equal (env-lookup x env)
                       (unit-set (mk-di-const tm)))
                (w-s s env))
           (equal (apply-function (e-s s store) x)
                  (apply-function store x))))

(disable e-s)
(disable w-s)

(prove-lemma stm-list-cant-clobber-const (rewrite)
  (implies (and (equal (env-lookup x env)
                       (unit-set (mk-di-const tm)))
                (w-s-list s-list env))
           (equal (apply-function (e-s-list s-list store) x)
                  (apply-function store x)))
  ((induct (e-s-list s-list store))
   (use (stm-cant-clobber-const (s (car s-list))))))
```

PART IV: THE LISP DEFINITION

Michael K. Smith

Chapter IV-1

SYNTAX: THE GRAMMAR

The syntax of these grammar rules should be obvious except for the constructors, which follow the symbol "==". (Note that as a typing convenience we have used "-" rather than "_" to separate the component parts of non-terminal names.) A vertical bar separates alternative realizations of a non-terminal.

The form following "==" is a Lisp form that describes how to compute a representation for the non-terminal out of its subcomponents. The dollar-signed integers are variables such that $i is bound to the representation of ith element of the right hand side of the production. An object declaration (see below) has, according to its first rule, 6 components. Some of these are syntactic sugar of no interest to us

[ARM (3.2)]

```
object-declaration ::=
   identifier-list : type-mark := expression ;        == ‘(<VARIABLE-DECL> ,$1 ,$3 ,$5)
 | identifier-list : CONSTANT type-mark := expression ;
                                                       == ‘(<CONSTANT-DECL> ,$1 ,$4 ,$6)
```

e.g. the colon, semi-colon, and assignment operator. We build a list with the following components: the first element (normally the operator) indicates what sort of form we have, the second is an identifier list of the objects to be declared, the third is the type of these objects, and the fourth is the initial value of the objects. Thus,

```
a,b : integer := x;
```

would result in the form:

```
(<VARIABLE-DECL> (<ID-LIST> A B) (<TYPE-MARK> INTEGER) X)
```

We are using the Common Lisp [Steele 84] backquote notation extensively, since most of these constructions are so simple. The backquoted form evaluates to a result that is identical to the original form, but with all expressions preceded by a comma replaced by the result of evaluating them. The ",@" prefix works similarly to the "," prefix within a backquote, except that the resulting list is spliced into place. Suppose X = 1.

```
‘(TEST ,(LIST X) Z)  => (TEST (1) Z)
‘(TEST ,@(LIST X) Z) => (TEST 1 Z)
```

The grammar depends on the lexical scanner to some extent, in that the scanner distinguishes reserved words, identifiers, special symbols, and literals. In particular, it is important that while nAVA does not use the reserved word **task**, it nonetheless prevents the user from using it as an identifier.

IV-1.1 Syntactic Rules

The form of these rules should be fairly clear. The form is:

[ARM (3.2)]
non-terminal ::= [Defined: 1, Static: 24, Dynamic 35]
pattern$_1$ == (function$_1$ \$1 \$3 \$5)
| pattern$_2$ == (function$_2$ \$1 \$2)
...
| pattern$_n$ == (function$_n$ \$1 \$4 \$6)

"[ARM (3.2)]" points to the location of the corresponding Ada construct in the Ada Language Reference Manual. The cross reference we have discussed previously (see page 3). It indicates that the non-terminal is defined informally on page 1 of the reference manual, its static semantics can be found on page 24, and its dynamic semantics on page 35.

Each pattern after "::=" is one realization of the non-terminal. A pattern is a list of non-terminals, terminals (in nanoAVA, just identifiers), reserved words and/or special-symbols. The "==" indicates the constructor for a branch of the rule. This constructor takes as arguments the values produced (recursively) for each element of the pattern (indicated by the positional variables $1, $2, etc.). Identifiers, reserved words and special-symbols just return their value (as a symbol). We will use the term *operator* to refer to the first element (CAR) of these lists and *arguments* to refer to the rest of the list (CDR).

```
                                                                          [ARM (3.2)]
object-declaration  ::=

     identifier-list : type-mark := expression ;
                                             == ‘(<VARIABLE-DECL> ,$1 ,$3 ,$5)
   | identifier-list : CONSTANT type-mark := expression ;
                                             == ‘(<CONSTANT-DECL> ,$1 ,$4 ,$6)
                                                                          [ARM (3.2)]
identifier-list  ::=                                                     [Defined: 20]
     identifier                              == ‘(<ID-LIST> ,$1)
   | identifier , identifier-list            == ‘(<ID-LIST> ,$1 ,@(CDR $3))
                                                                        [ARM (3.3.2) ]
type-mark ::=                                                 [Defined: 21,Static: 76]
     name                                    ==‘(<TYPE-MARK> ,$1)
                                                                          [ARM (3.9)]
declarative-part ::=                          [ARM: 23,Deno: 49,Static: 76,Dynamic: 81]
     basic-decls                             == ‘(<DECLARATIVE-PART> ,@$1)

basic-decls  ::=

     basic-declarative-item                  == (LIST $1)
   | basic-declarative-item basic-decls      == (CONS $1 $2)
                                                                         [ARM (3.9) ]
basic-declarative-item  ::=                   [ARM: 23,Deno: 49,Static: 76,Dynamic: 81]
     object-declaration                      == $1
                                                                          [ARM (4.1)]
name  ::=                                                                [Defined: 25]
     simple-name                             == $1

simple-name  ::=                                                          [ARM (4.1)]
     identifier                              == $1
```

```
                                                                        [ARM (4.4)]
expression ::=                                  [ARM: 25,Deno: 49,Static: 78,Dynamic: 81]
     relation                          == $1
                                                                        [ARM (4.4)]
relation ::=                                    [ARM: 25,Deno: 49,Static: 78,Dynamic: 81]
     simple-expression                 == $1
    | simple-expression relational-operator simple-expression
                                       == (LIST $2 $1 $3)

simple-expression ::=     term         == $1

term ::=                                                                [ARM (4.4)]
     factor                            == $1

factor ::=                                                              [ARM (4.4)]
     primary                           == $1

primary ::=                                                             [ARM (4.4)]
     name                              == $1
                                                                        [ARM (4.5)]
relational-operator ::=                         [ARM: 26,Deno: 49,Static: 78,Dynamic: 81]
     =                                 == '<EQUAL>
    | /=                               == '<NE>
    | <                                == '<LT>
    | <=                               == '<LE>
    | >                                == '<GT>
    | >=                               == '<GE>
                                                                        [ARM (5.1)]
sequence-of-statements ::=                      [ARM: 29,Deno: 51,Static: 77,Dynamic: 81]
     statements                        == `(<SEQ-OF-STMTS> ,@$1)

statements ::=

     statement                         == (LIST $1)
    | statement statements             == (CONS $1 $2)
                                                                        [ARM (5.1)]
statement ::=                                   [ARM: 29,Deno: 49,Static: 77,Dynamic: 81]
     simple-statement                  == $1

simple-statement ::=                                                    [ARM (5.1)]
     null-statement                    == $1
    | assignment-statement             == $1
    ;; Documentation for Main Procedure invocation.
    ;; Not contained in the nanoAVA grammar.
    ;; | procedure-call-statement      == $1
                                                                        [ARM (5.1)]
null-statement ::=                              [ARM: 29,Deno: 52,Static: 77,Dynamic: 81]
     NULL ;                            == '(<NULL-STMT>)
                                                                        [ARM (5.2)]
assignment-statement ::=                        [ARM: 29,Deno: 52,Static: 77,Dynamic: 81]
     name := expression ;              == `(<ASSIGN-STMT> ,$1 ,$3)
                                                                        [ARM (6.1)]
subprogram-specification ::=                             [ARM: 31,Static: 76,Dynamic: 81]
     PROCEDURE identifier              == `(<PROCEDURE-SPEC> ,$2 NIL)
    | PROCEDURE identifier formal-part == `(<PROCEDURE-SPEC> ,$2 ,$3)

formal-part ::=                                                         [ARM (6.1)]
     ( formal-part2                    == `(<FORMAL-PART> ,@$2)
formal-part2  ::=

        parameter-specification )                    == (LIST $1)
      | parameter-specification ; formal-part2       == (CONS $1 $3)

;; No default value for parameters
                                                                                [ARM (6.1)]
parameter-specification  ::=                                                    [Defined: 31]
        identifier-list : mode type-mark             == `(<PARM-SPEC> ,$1 ,$3 ,$4)
                                                                                [ARM (6.1)]
mode  ::=                                                           [Defined: 31,Static: 76]
        IN OUT                                       == '(<MODE> *IN-OUT*)
                                                                                [ARM (6.3)]
subprogram-body  ::=                            [ARM: 32,Deno: 49,Static: 76,Dynamic: 81]
        subprogram-specification sub-decl
          BEGIN sequence-of-statements END ;         == `(,@$1 ,$2 ,$4)

sub-decl  ::=

        IS                                           == '(<DECLARATIVE-PART> NIL)
      | IS declarative-part                          == $2

compilation  ::=                                [ARM: 41,Deno: 49,Static: 75,Dynamic: 81]
        compilation-unit                             == `(<COMPILATION> ,$1)
                                                                               [ARM (10.1)]
compilation-unit  ::=                           [ARM: 41,Deno: 49,Static: 75,Dynamic: 81]
        library-unit                                 == `(<COMPILATION-UNIT> ,$1)
                                                                               [ARM (10.1)]
library-unit  ::=                                       [ARM: 41,Static: 75,Dynamic: 81]
        subprogram-body                              == $1
```

The procedure call description below is provided so that we have somewhere to stand when describing the invocation of the main program. These productions are not part of the nanoAVA grammar.

```
procedure-call-statement  ::=                                                   [ARM (6.4)]
        name ;                                       == `(<PROCEDURE-CALL-STMT> ,$1 NIL)
      | name actual-parameter-part ;                 == `(<PROCEDURE-CALL-STMT> ,$1 ,$2)

actual-parameter-part  ::=                                                      [ARM (6.4)]
        ( actual-parameter-part2                     == `(<ACTUAL-PARAMETER-PART> ,@$2)

actual-parameter-part2  ::=

        parameter-association )
                                                     == (LIST $1)
      | parameter-association , actual-parameter-part2
                                                     == (CONS $1 $3)

parameter-association  ::=                                                      [ARM (6.4)]
        actual-parameter                             == `(<PARM-ASSOC> ,$1)

actual-parameter  ::=                                                           [ARM (6.4)]
        expression                                   == $1
```

IV-1.2 Syntactic Output

The result of a successful syntactic pass is a compilation, whose structure is described below. A "!" indicates a list of elements. The form (A . B) indicates that A is to be CONSed onto the list B. That is, A is the first element of the list and B contains the rest of the list.

```
compilation  =
                  (<compilation>
                   (<compilation-unit>
                    (<procedure-spec> identifier
                     (<formal-part> . parameter-specification!)
                     (<declarative-part> . object-declaration!)
                     (<seq-of-stmts> . simple-statement!))))

parameter-specification  =
                  (<parm-spec> identifier-list (<mode> *in-out*) type)

type =
                  (<type-mark> identifier)

object-declaration  =
                  (<variable-decl> identifier-list type expression)
                | (<constant-decl> identifier-list type expression)

identifier-list  =
                  (<id-list> . identifer!)

expression  =
                  identifier
                | (<EQUAL> identifier identifier)  |  (<NE> identifier identifier)
                | (<LT> identifier identifier)     |  (<LE> identifier identifier)
                | (<GT> identifier identifier)     |  (<GE> identifier identifier)

simple-statement  =
                  (<null-stmt>)
                | (<assign-stmt> identifier expression)
```

The procedure call description below is provided so that we can talk about the invocation of the main program. But it will not be produced by the parser and we do not need to provide static semantic routines to check it.

```
simple-statement =
   ;; previous simple-statement plus ...
   (<procedure-call-stmt> identifier
                         (<actual-parameter-part> . parameter-association!))

parameter-association  =  (<parm-assoc> expression)
```

Chapter IV-2
STATIC SEMANTICS

This section presents the complete functional description of the static semantic checks performed over the syntactic output. Two basic definitional approaches are used. We define a number of Lisp functions (using DEFUN) and we define a set of semantic analysis routines (using DEFSEMANTICS) that key off of the operators in the syntactic output.

The entry point to static semantic checking is the function *semantic-check-and-convert*. It takes as argument the syntactically analyzed program. The function normalizes the input, checks that **program** is a compilation and then checks the static semantics of the program by applying the function **WF**. If semantically ok, it returns the transformed internal form to pass on to the dynamic semantics.

This input form is the *program* to be passed to **INTERPRET**. (For more detail on this process and its supporting data structures see page 81).

The `defsemantics` form is used to define semantic checks and transformations for the syntactic output. The form of a call to defsemantics is:

```
    (defsemantics prefix-template
       [ Let      v1 = value1
         [ [and] vn = valuen ] ]
       [ WF        = [ test { string } ]+  ]
       [ Transform  = sexpr ]
       [ Declare    = sexpr ]
       [ Normalize  = sexpr ])

    prefix-template ::= (opr arg_1 ... arg_n) | (opr &repeating args)
```

The prefix-template is a syntactic type descriptor. Its car is the type name (or operator) and its cdr is a list of argument names. There are two basic forms of prefix, one with a fixed number of arguments, each of which may be of a different type, and the second with an unspecified number of arguments of a homogeneous type. The "&repeating" keyword supports the later form. Contrast

```
       (<procedure-spec> swap formal-part decl-part seq-of-stmts)
    vs.
       (<id-list> x y z)
```

Defsemantics creates a set of functions that are associated with prefix forms headed by **opr**. When evaluating the forms in the definition the following additional bindings will be in effect:

- ENV - the environment in which the semantics of the form are being evaluated.
- OPR - the operator name.
- FORM - the entire prefix form being analyzed.
- arg_n - for each named argument of the prefix template, the corresponding element of the form being evaluated.

Each of the clauses (LET, WF, TRANSFORM, DECLARE, NORMALIZE) are optional.

LET introduces some local variables and their bindings, just to clean up the definition textually, if the user wishes.

WF introduces a sequence of predicates and optional associated error message. The convention is that a form is well formed if all of the predicates evaluate to true. If a test fails and it has an associated string, the string constitutes its error message.

TRANSFORM provides a means to convert the input form, perhaps without overloading computed, into an alternative form containing more semantic information.

DECLARE states how this form modifies the environment. The sexpr returns an environment.

The above keywords are also the names of functions which apply the corresponding operation to a piece of prefix.

If a component fails its semantic check then prefix of the form `(*semantic-error* form reason)` should be returned. (This allows us to continue applying semantic checks after we have encountered a first error, in an effort to wring as much information out of the input as we can. This is peripheral to the determination of what the dynamic semantics accepts, but the capability for evaluation provides a useful check on the adequacy of our definition.)

This modified version of the Lisp form is based in large part on the denotational definition of nanoAVA (see 47).

IV-2.1 Static Semantics Entry Point

```
;; Uses support in denotation.lisp.
;;
;; Tue May 24 19:23:37 1988, by MKSmith

(defun semantic-check-and-convert (program)
  ;; Takes the result of the syntactic pass, PROGRAM, checks its
  ;;  static semantics, and converts to interpreter input form.  The
  ;;  value returned by this function is the LIBRARY to be passed to
  ;;  AUGMENT-AND-RUN.
  ;;
  ;; First we normalize, then make sure we are dealing with a full
  ;;  syntactic compilation.
  ;; Elsewhere we check to ensure that no programs containing
  ;;  syntactic errors are passed to this function.
  ;;
  (let ((np (normalize program)))
    (cond ((not (eqopr np '<compilation>))
           (semantic-error "Program is not a compilation" np standard-env))
          ((wf np standard-env)
           (transform np standard-env))
          (t (format t "~%Semantic errors in program.~%")
             (transform np standard-env)))))


;; ENV (the environment) is a pair of a set of local names and an
;; alist of declarations.

(defvar standard-env
  '((INTEGER BOOLEAN FLOAT CHARACTER
             ASCII NATURAL POSITIVE STRING DURATION
             CONSTRAINT_ERROR NUMERIC_ERROR PROGRAM_ERROR
             STORAGE_ERROR TASKING_ERROR
             TRUE FALSE)
    ((integer *type*)
     (boolean *type*))))
```

IV-2.2 Basic Support Function Definitions

```
(defun locals (env)       (car env))
(defun M (env)            (cadr env))

;; The declarations are stored in an alist.  Each entry is of the
;; form:
;;         (name . declarative-info)
;;
;; There may not be more than one element in the cdr of the list that
;; contains declarative info.  This will change with AVA.
;;
;; Declarative info may be one of:
;;  (*var* typename) (*const* typename) (*proc*) (*type*)

(defun lookup (id env)
  (cdr (assoc id (M env))))

(defun hide-homograph (name env)
  ;; The meaning of the name is changed, to undefined.
  (list (locals env) (cons (cons name nil) (M env))))

(defun start-declarative-region (env)
  (list nil (M env)))

(defun set-decl (id decl env)
  ;; The id get decl as its meaning.  Overwrites.  This seems to be
  ;; ok for nanoAVA.
  (list (enter id (locals env))
        (cons (cons id decl) (M env))))

(defun kind-in-env (id env)
  ;; Elements of env are
  ;;  (*var* typename) or (*const* typename) or (*proc*) or (*type*)
  ;; Assumes (lookup id env)
  (opr (lookup id env)))

(defun type-in-env (id env)
  ;; Assumes (lookup id env)
  ;; May return NIL for (*proc*) and (*type*).
  (arg1 (lookup id env)))
```

IV-2.3 Compilation

```
  (defsemantics (<compilation> unit)            [ARM: 41,Deno: 49,Syntax: 70,Dynamic: 81]

  WF        = (WF unit env)
  Transform = (Transform unit env))

 (defsemantics (<compilation-unit> unit)        [ARM: 41,Deno: 49,Syntax: 70,Dynamic: 81]

  WF        = (WF unit env)
  Transform = (Transform unit env))


;; Note that there is no reason to declare a procedure.  In particular
;; there is no reason to add the procedure declaration to env.  It can never be used.
;; It can be replaced in the environment by a formal or local declaration, but
;; there is no need to distinguish this replacement from the declaration of a variable
;; that does not have the same name as the procedure.

  (defsemantics (<procedure-spec> name formal-part decl-part seq-of-stmts)

  LET  env1 = (start-declarative-region env)

  WF        = (not (member name (locals env)))
                  "Redefining STANDARD predefined name"
              (WF formal-part  env1)
              (WF decl-part    (declare formal-part env1))
              (WF seq-of-stmts (declare decl-part (declare formal-part env1)))
                                                [ARM: 32,Deno: 49,Syntax: 70,Dynamic: 81]
  Transform = (prefix '*procedure-spec* name
                      (transform formal-part env1)
                      (transform decl-part    (declare formal-part env1))
                      (transform seq-of-stmts
                                 (declare decl-part (declare formal-part env1)))))


 (defsemantics (<formal-part> &repeating parameters)       [ARM: 31,Syntax: 69,Dynamic: 81]
  Normalize = (prefix '<formal-part> (mapcan #'normalize-parm-spec parameters))
  WF        = (WF-map parameters env)
  Declare   = (declare-map (args (transform form env)) env)
  Transform = (prefix '*formal-part* (transform-map parameters env)))


  (defsemantics (<single-parm-spec> id type)                                    [Static: 76]
  ;; Mode is gone
  ;; Type is  (<type-mark> typename)

  WF        = (not (member id (locals env)))
                  "Attempt to redefine predefined unit or local name"
              (WF type (hide-homograph id env))

  Declare   = (set-decl id (list '*VAR* (arg1 type)) env)
  Transform = (prefix '*single-parm-spec* id (transform type env)))

#|  Naming and Visibility: some examples.

     procedure foo ( foo : T ) is      ... OK, procedure foo hides homographs.
     procedure float (x : boolean) is  ... NO, can't use predefined identifiers for main.
     float : boolean := x;    OK, not in same local declarative region.
     begin
      x : boolean := foo;
      x : integer := bar; ... Can't redefine within same declarative region.  |#
```

IV-2.4 Types and Declarations

```
;; Already checked mode in syntax.

 (defsemantics (<mode> mode)                                       [Syntax: 70,Static: 76]


  WF        = TRUE
  Transform = (prefix '*mode* (list mode)))

 (defsemantics (<type-mark> id)
                                                         [Deno: 49,Syntax: 68,Static: 76]
  WF        = (member id (locals env))           "Unknown type"
              (equal (lookup id env) '(*TYPE*))    "Not a type"

  Transform = id)

  (defsemantics (<declarative-part> &repeating objects)
                                                [ARM: 23,Deno: 49,Syntax: 68,Dynamic: 81]
  Normalize = (prefix '<declarative-part> (mapcan #'normalize-decl objects))

  WF        = (WF-map objects env)
  Transform = (prefix '*declarative-part* (transform-map objects env))
  Declare   = (declare-map (args (transform form env)) env))


 (defsemantics (<single-variable-decl> id type value)

  WF        = (not (member id (locals env)))
                "Redefining predefined unit or local name"
              (WF type (hide-homograph id env))    ;; ??
              (WF-expression value type (hide-homograph id env))

  Transform = (prefix '*variable-decl* id
                      (transform type env)
                      (transform-expression value type env))
  Declare   = (set-decl id (list '*VAR* (arg1 type)) env))


 (defsemantics (<single-constant-decl> id type value)

  WF        = (not (member id (locals env)))
                  "Attempt to redefine predefined unit or local name"
              (WF type (hide-homograph id env))    ;; ??
              (WF-expression value type (hide-homograph id env))

  Transform = (prefix '*constant-decl* id
                      (transform type env)
                      (transform-expression value type env))
  Declare   = (set-decl id (list '*CONST* (arg1 type)) env))
```

IV-2.5 Statements

```
  (defsemantics (<seq-of-stmts> &repeating statements)  [ARM: 29,Deno: 51,Syntax: 69,Dynamic: 81]

 WF        = (WF-map statements env)
 Transform = (prefix '*seq-of-stmts* (transform-map statements env)))


(defsemantics (<null-stmt>)                   [ARM: 29,Deno: 52,Syntax: 69,Dynamic: 81]

 WF        = true
 Transform = (prefix '*null-stmt* nil))


(defsemantics (<assign-stmt> id value)           [ARM: 29,Deno: 52,Syntax: 69,Dynamic: 81]

 WF        = (equal (kind-in-env id env) '*VAR*)
                "Assignment can only be to a variable"
             (wf-expression value (type-in-env id env) env)
                "Value ill-formed"

 Transform = (prefix '*assign-stmt*
                     (list id (transform-expression value (type-in-env id env) env))))
```

IV-2.6 Expressions

```
(defun wf-expression (e type env)                  [ARM: 25,Deno: 49,Syntax: 69,Dynamic: 81]
  (if (identifier-p e)
      (and (member e (locals env))
           (member (kind-in-env e env) '(*VAR* *CONST*))
           (equal  (type-in-env e env) type))
      (if (lookup (arg1 e) env)
          (let ((type2 (type-in-env (arg1 e) env)))
            (and (equal type 'boolean)
                 (wf-expression (arg1 e) type2 env)
                 (wf-expression (arg2 e) type2 env)))
          FALSE)))

(defun transform-expression (e t env)
  (if (identifier-p e)
      e
      (prefix (unload-type (opr e) (type-in-env (arg1 e) env))
              (arg1 e) (arg2 e))))

(defun unload-type (op type)                       [ARM: 26,Deno: 49,Syntax: 69,Dynamic: 81]
  (if (equal type 'integer)
      (case op
            (<equal> '*equal-int*)
            (<ne>    '*neq-int*)
            (<lt>    '*lt-int*)
            (<le>    '*le-int*)
            (<gt>    '*gt-int*)
            (<ge>    '*ge-int*))
      ;; Else must be boolean.
      (case op
            (<equal> '*equal-bool*)
            (<ne>    '*neq-bool*)
            (<lt>    '*lt-bool*)
            (<le>    '*le-bool*)
            (<gt>    '*gt-bool*)
            (<ge>    '*ge-bool*))))
```

IV-2.7 Normalization Functions

```
;; Unwind multiple declarations.
;;
;;   (<variable-decl> identifier-list (<type-mark> identifier) expression)
;;  becomes
;;   ((<single-variable-decl> identifier (<type-mark> identifier) expression)*)
;;
;; And similarly for <constant-decl>.


(defun normalize-decl (form)
  (if (eqopr form '<variable-decl>)
      (normalize-variable-decl form)
      (normalize-constant-decl form)))

(defun normalize-variable-decl (form)
  ;;  (<variable-decl> ids type value)
  (let ((ids (arg1 form))
        (type (arg2 form))
        (value (arg3 form)))
    (mapcar (f-l (x)
                 (prefix '<single-variable-decl> (list x type value)))
            ids)))
```

```
(defun normalize-constant-decl (form)
  ;;  (<constant-decl> ids type value)
  (let ((ids (arg1 form))
        (type (arg2 form))
        (value (arg3 form)))
    (mapcar (f-l (x)
                 (prefix '<single-constant-decl> (list x type value)))
            ids)))

;; Do the same for <parm-spec>s.
;;
;;   (<parm-spec> identifier-list (<mode> *in-out*) (<type-mark> identifier))
;;  becomes
;;   ((<single-parm-spec> identifier (<type-mark> identifier))*)

(defun normalize-parm-spec (form)
  ;;  (<parm-spec> ids mode type)
  (let ((ids (arg1 form))
        (mode (arg2 form))
        (type (arg3 form)))
    (mapcar (f-l (x)
                 (prefix '<single-parm-spec> (list x type)))
            ids)))
```

IV-2.8 Static Semantic Output

We have taken a slight liberty with parentheses in the interest of readability. The actual internal form looks like

(opr (arg$_1$ ... arg$_n$) (annotation$_1$ .. annotation$_m$))

Which we write here as

(opr arg$_1$ ... arg$_n$ {annotation$_1$ .. annotation$_m$})

In this version there will be no annotations.

After static semantic analysis the internal form of a nAVA program is as follows:

```
program ::=
                procedure-spec

procedure-spec  ::=
                (*PROCEDURE-SPEC* identifier
                                (*FORMAL-PART* . parameter-specification!)
                                (*DECLARATIVE-PART* . object-declaration!)
                                (*SEQ-OF-STMTS* . simple-statement!))

parameter-specification  ::=
                (*SINGLE-PARM-SPEC* identifier type)

object-declaration  ::=
                (*VARIABLE-DECL* identifier type expression)
                (*CONSTANT-DECL* identifier type expression)

expression  ::=
                simple-expression
                (op simple-expression simple-expression)

op  ::=
                *EQUAL-INT* | *EQUAL-BOOL* |
                *NEQ-INT*   | *NEQ-BOOL*   |
                *LT-INT*    | *LT-BOOL*    |
                *LE-INT*    | *LE-BOOL*    |
                *GT-INT*    | *GT-BOOL*    |
                *GE-INT*    | *GE-BOOL*

simple-expression ::=
               identifier | TRUE | FALSE | number

simple-statement  ::=
                (*NULL-STMT*)
                (*ASSIGN-STMT* identifier expression)
```

Programs containing semantic errors will contain occurences of an error form. The FORM argument is the internal representation that was under analysis when the error was detected. The string should contain a description of the error.

```
error-form ::= (*SEMANTIC-ERROR* form string)
```

The integer and boolean prefix values below are added to expressions so that the primitive operations can be defined over literals. This also allows the calling environment of the main program to provide actual values for main parameters.

```
expression ::=
                ;; previous expression plus
                FALSE | TRUE | integer
```

Chapter IV-3

DYNAMIC SEMANTICS: THE INTERPRETER DEFINITION

IV-3.1 Entry Point

The initial state is a list of variable bindings of the form (variable-name . value). `Run-program` intreprets the program form in `initial-state`.

```
(defvar initial-state nil)
(defvar program nil)

(defun run-program (program)
  (interpret program initial-state))

;;; BASIC INTERPRETER ENTRY POINT


(defun INTERPRET (x s)
  ;; X is a statement, S is a state.
  ;;
  (case (opr x)
        (*PROCEDURE-SPEC*                       [ARM: 32,Deno: 49,Syntax: 70,Static: 76]
         (interpret (get-stmts x) (interpret (get-decls x) s)))
        (*DECLARATIVE-PART*                     [ARM: 23,Deno: 49,Syntax: 68,Static: 76]
         (if (null (args x)) s
             (Interpret (rest-of-decls x)
                        (Interpret (arg1 x) s))))
        (*CONSTANT-DECL*           (set-value (arg1 x) (evaluate (arg3 x) s) s))
        (*VARIABLE-DECL*          (set-value (arg1 x) (evaluate (arg3 x) s) s))
        (*SEQ-OF-STMTS*                         [ARM: 29,Deno: 51,Syntax: 69,Static: 77]
         (if (null (args x)) s
             (Interpret (rest-of-stmts x)
                        (Interpret (arg1 x) s))))

        (*NULL-STMT* s)                         [ARM: 29,Deno: 52,Syntax: 69,Static: 77]
        (*ASSIGN-STMT*                          [ARM: 29,Deno: 52,Syntax: 69,Static: 77]
         (set-value (arg1 x) (evaluate (arg2 x) s) s))))


 (defun evaluate (exp s)                        [ARM: 25,Deno: 49,Syntax: 69,Static: 78]
  (if (identifier-p exp)
      (get-value exp s)
      (eval-fun (opr exp)
                (evaluate (arg1 exp) s)
                (evaluate (arg2 exp) s))))

  (defun eval-fun (opr x y)
  (case opr
        (*equal-int* (if (equal x y) true false))
        (*neq-int*   (if (not (equal x y)) true false))
        (*lt-int*    (if (lt x y) true false))
        (*le-int*    (if (le x y) true false))
        (*gt-int*    (if (gt x y) true false))
        (*ge-int*    (if (ge x y) true false))
        (*equal-bool* (if (equal x y) true false))
        (*neq-bool*   (if (not (equal x y)) true false))
        (*lt-bool*    (if (and (equal x false) (equal y true))  true false))
        (*le-bool*    (if (or  (equal x false) (equal y true))  true false))
        (*gt-bool*    (if (and (equal x true)  (equal y false)) true false))
        (*ge-bool*    (if (or  (equal x true)  (equal y false)) true false))))


(defun rest-of-stmts (stmtlist)
  (if (args stmtlist)
      (prefix '*SEQ-OF-STMTS* (cdr (args stmtlist)))
      (prefix '*SEQ-OF-STMTS* nil)))

(defun rest-of-decls (decllist)
  (if (args decllist)
      (prefix '*DECLARATIVE-PART* (cdr (args decllist)))
      (prefix '*DECLARATIVE-PART* nil)))

(defun get-decls (proc) (arg3 proc))

(defun get-stmts (proc) (arg4 proc))


(defun normal-state (s)
  (not (equal s 'abnormal-state)))

(defun set-value (id value s)
  (cons (cons id value) s))

(defun get-value (id s)
  (cdr (assoc id s)))


;; Example

(setq initial-state '((a . 1) (b . 2)))

(setq program
      '(*procedure-spec* swap
                         (*formal-part*
                          (*single-parm-spec* x integer)
                          (*single-parm-spec* y integer))
                         (*declarative-part*
                          (*constant-decl* temp integer x))
                         (*seq-of-stmts*
                          (*assign-stmt* x y)
                          (*assign-stmt* y temp))))
```

Chapter IV-4

UTILITY AND DENOTATIONAL SUPPORT FUNCTIONS

```
;;; Denotational Definition Support.

;;; This file provides a harness for defining and evaluating a
;;; denotational style of static semantics in Lisp.
;;; Effort inspired by Mark Saaltink's denotational nanoAVA
;;; definition.

;;; Implementation dialect: Common Lisp
;;; Fri May 27 17:33:35 1988, by MKSmith

(defmacro defsemantics (form &rest arguments)
  ;;
  ;; (defsemantics (opr . args)
  ;;   [ Let    v1 = value1
  ;;    [ [and] vn = valuen ] ]
  ;;   [ WF        = [ test { string } ]+  ]
  ;;   [ Transform = sexpr ]
  ;;   [ Declare   = sexpr ]
  ;;   [ Normalize = sexpr ])

  ;; FORM is a syntactic type descriptor.  Its car is the type name
  ;; (or operator) and its cdr is a list of argument names.  When
  ;; evaluating the forms in the definition the following bindings
  ;; will be in effect:

  ;;    ENV  - the environment in which the semantics of the form are
  ;;           being evaluated
  ;;    OPR  - the operator name
  ;;    argn - for each named element of args, the corresponding
  ;;           element of the form being evaluated.

  ;; Each of the clauses (LET, WF, TRANSFORM, DECLARE, NORMALIZE) are
  ;; optional.

  ;; LET introduces some local variables and their bindings, just to
  ;; clean up the definition textually, if the user wishes.

  ;; WF introduces a sequence of predicates and optional associated
  ;; error message.  The convention is that a form is well formed if
  ;; all of the predicates evaluate to true.  If a test fails and it
  ;; has an associated string, the string constitutes its error
  ;; message.

  ;; TRANSFORM provides a means to convert the input form, perhaps
  ;; without overloading computed, into an alternative form containing
  ;; more semantic information.  The variable TRANSFORM is bound to
  ;; the result of evaluating sexpr.

  ;; DECLARE states how this form modifies the environment.  The sexpr
  ;; returns an environment.
  ;;
  `(let ((opr (opr ,form))
                                        ; returns a list of binding pairs
         (let-clauses      (extract-let ,arguments opr))
                                        ; returns a COND
         (wf-clauses       (extract-wf ,arguments opr))
         (normalize-clause (extract-normalize ,arguments opr))
         (transform-clause (extract-transform ,arguments opr))
         (declare-clause   (extract-transform ,arguments opr)))
     ;;
     (if let-clauses
         (put opr 'let-variables      let-clauses))
     ;;
     (if wf-clauses
         (put opr 'wf-function        (build-function form wf-clauses)))
     ;;
     (if normalize-clause
         (put opr 'normalize-function (build-function form normalize-clause)))
     ;;
     (if transform-clause
         (put opr 'transform-function (build-function form transform-clause)))
     ;;
     (if declare-clause
         (put opr 'declare-function   (build-function form declare-clause)))))

(defun extract-wf (l name)
  (let ((wf (cdr (member 'wf l))))
    (if (and wf (equal (car wf) '=)) (setq wf (cdr wf)))
    (if wf (cons 'cond (build-wf-cond wf name)))))

(defun build-wf-cond (wf name)
  (cond ((null wf) (cons '(t true) nil))
        ((member (car wf) '(LET TRANSFORM DECLARE NORMALIZE)) (cons '(t true) nil))
        ((equal (car wf) '=) (format t "~%Error in WF component of ~A" name) nil)
        ((stringp (cadr wf))
         (cons (list (list 'not (car wf)) `(semantic-error form ,(cadr wf)))
               (build-wf-cond (cddr wf) name)))
        (t (cons (list (list 'not (car wf)) `(semantic-error form "Error"))
                 (build-wf-cond (cdr wf) name)))))

;; Semantic Error

(defvar *semantic-error* '*semantic-error*)

(defun semantic-error (msg form)
  (prefix *semantic-error* `(,form ,msg)))


(defun extract-let (l name)
  (let ((let (cdr (member 'let l))))
    (if (and let (equal (car let) '=)) (setq let (cdr let)))
    (if let (build-let let name))))

(defun build-let (let name)
  (cond ((null let) nil)
        ((member (car let) '(TRANSFORM DECLARE NORMALIZE)) nil)
        ((equal (car let) '=) (format t "~%Error in LET component of ~A" name) nil)
        ((equal (car let) 'and) (build-let (cdr let) name))
        ((equal (car let) 'in)  (build-let (cdr let) name))
        ((symbolp (car let))
         (cons (list (car let) (extract-let-value (cdr let)))
               (build-let (after-let-value let) name)))
        (t (format t "~%Error in LET component of ~A" name) nil)))

(defun extract-let-value (let-tail)
  (cond ((equal (car let-tail) '=)
         (cadr let-tail))
        (t (format t "~%Error in let component of ~A" name) nil)))
```

```
(defun after-let-value (let-tail)
  (cond ((equal (car let-tail) '=)
         (cddr let-tail))
        (t (format t "~%Error in let component of ~A" name) nil)))

(defun extract-normalize (l name)
  (let ((rest (cdr (member 'normalize l))))
    (if (and rest (equal (car rest) '=)) (setq rest (cdr rest)))
    (cond ((null rest))
          ((or (null (cdr rest)) (member (cadr rest) '(let wf transform declare)))
           (car rest))
          (t (format t "~%Error in normalize component of ~A" name) nil))))

(defun extract-transform (l name)
  (let ((rest (cdr (member 'transform l))))
    (if (and rest (equal (car rest) '=)) (setq rest (cdr rest)))
    (cond ((null rest))
          ((or (null (cdr rest)) (member (cadr rest) '(let wf normalize declare)))
           (car rest))
          (t (format t "~%Error in transform component of ~A" name) nil))))

(defun extract-declare (l name)
  (let ((rest (cdr (member 'declare l))))
    (if (and rest (equal (car rest) '=)) (setq rest (cdr rest)))
    (cond ((null rest))
          ((or (null (cdr rest))
               (member (cadr rest) '(let wf transform normalize)))
           (car rest))
          (t (format t "~%Error in declare component of ~A" name) nil))))
```

```
;; In order to apply the WF, Transform, or Declare just
;; call them on an object.  These functions will get the version
;; appropriate to the type of the expression and apply it.

(defun WF (expr env)
  ;;
  ;; For every type of expression that we apply WF to there should be
  ;; an associated test.  If not, our definition contains an error.
  ;; So we do not make the application of the function conditional on
  ;; it existing.  If it isn't there we break.
  ;;
  (static-trace-entry expr)
  (static-trace-exit (apply (WF-function expr) (list expr env)) expr))

(defun WF-map (expr-list env)
  (cond ((null expr) true)
        ((WF (car expr-list) env)
         (WF-map (cdr expr-list) (declare (car expr-list) env)))
        (t nil)))


(defun Transform (expr env)
  ;; If there is no transform, leave it alone.
  (let ((fun (Transform-function expr)))
    (if fun
        (apply fun (list expr env))
        expr)))

(defun Transform-map (expr env)
  (if (null expr)
      nil
      (cons (transform (car expr-list) env)
            (transform-map (cdr expr-list) (declare (car expr-list) env)))))


(defun Declare (expr env)
  ;;
  ;; If there is no declare, return env.  We do this because declare
  ;; is applied in places where we don't know if one has been defined
  ;; for the type, e.g. the -map functions just above.
  ;;
  (let ((fun (Declare-function expr)))
    (if fun
        (apply fun (list expr env))
        env)))

(defun Declare-map (expr env)
  (if (null expr)
      env
      (declare-map (cdr expr-list) (declare (car expr-list) env))))


;; Normalize works a differently from the others.  It is meant to be
;; done before anything else.  The entire program should be normalized
;; before any other operation.  E.g. (NORMALIZE program).

(defun normalize (expr)
  (if (opr expr)
      (let ((norm (normalize-function (opr expr))))
        (if norm
            (apply norm (list expr))
            (prefix (opr expr) (mapcar #'normalize (args expr)))))
      expr))


(defun WF-function (expr)        (get (opr expr) 'wf-function))
(defun Transform-function (expr) (get (opr expr) 'transform-function))
(defun Declare-function (expr)   (get (opr expr) 'declare-function))
(defun normalize-function (expr) (get (opr expr) 'normalize-function))
```

```
(defvar *arg-extraction-fns* '(arg1 arg2 arg3 arg4 arg5
                               arg6 arg7 arg8 arg9 arg10))


(defun build-function (form function)
  ;;
  ;; FORM is of the form (opr arg1 ... argn) or (opr &repeating args).
  ;;  E.g. (<assign-stmt> id expr) or (<parameter-list> &repeating parameters).
  ;;
  ;; We build a form like
  ;;  (function (lambda (form env)
  ;;       (let ((opr (opr form))
  ;;             (id (arg1 form))
  ;;             (expr (arg2 form)))
  ;;         function)))
  ;; And return it.  The caller stores it.
  ;;
  ;; The function we compute has the variables FORM (the whole
  ;;  expression), OPR (the operator), the argument names, and ENV (the
  ;;  environment). We also provide TRANSFORM as a variable bound to
  ;;  (Transform FORM).
  ;;;
  ;; First we bind the variables according of form.
  ;; The definition may have included LET variables.  They will be
  ;; stored as properties under the car of form.
  ;;
  `(function (lambda (form env)
               (let ((opr (opr yyy))
                     ,@(extract-args (cdr form) 'yyy))
                 (let* ,(get (car form) '*let-variables*)
                   (let ((transform (transform form)))
                     ,function))))))

(defun extract-args (args argname)
  ;; Args is either a list of atomic argument names or a repeating group
  ;; which is indicated by (&repeating list-parameter).
  (cond ((null args) nil)
        ((equal (car args) '&repeating)
         (list (list (cadr args) (list 'args argname))))
        (t (extract-args2 args argname))))

(defun extract-args2 (args argname)
  (mapcar (function (lambda (x argfn) `(,x (,argfn ,argname))))
          args *arg-extraction-fns*))

(defparameter *trace-static-indent* 1)

(defparameter *trace-static* nil)

(defun static-trace-entry (expr)
  (when *trace-static*
    (when (or (member 'all *trace-static*)
              (member (opr expr) *trace-static*))
      (format t "~%~VTEnter Static Semantic: ~A"
              *trace-static-indent* expr)
      (incf *trace-static-indent*)
      nil)))

(defun static-trace-exit (result expr)
  (when *trace-static*
    (if (or (member 'all *trace-static*)
            (member (opr expr) *trace-static*))
        (format t "~%~VTExit Static Semantic: ~A"
                (decf *trace-static-indent*) result)))
  result)

(defmacro static-trace (&rest l)
  `(let ((l2 ',l))
     (cond ((null l2) *trace-static*)
           ((consp (car l2))
            (setq *trace-static* (union (car l2) *trace-static*)))
           ((consp l2)
            (setq *trace-static* (union (list l2) *trace-static*)))
           (t (format t "~%Bad argument to static-trace.~%")))))

(defmacro static-untrace (&rest l)
  `(let ((l2 ',l))
     (cond ((null l2) *trace-static*)
           ((equal l2 '(all)) (setq *trace-static* nil))
           ((consp (car l2))
            (setq *trace-static* (set-difference *trace-static* (car l2))))
           (t (format t "~%Bad argument to static-trace.~%")))))
```

IV-4.1 Parser Utilities

```
;; Macros, especially for reading/printing with column location knowledge.

;; Module and Package manipulation

(in-package "PRS")

(export '(write-errors ll=line-errors)      ; (export symbols &optional package)
        (find-package "PRS"))


;; Completed module and package manipulation

(eval-when (load compile eval)

;; Basic.  Should be in init file.

(defmacro memq (a l) `(member ,a ,l :test #'eq))

(defmacro nconc1 (l a) `(nconc ,l (list ,a)))

;; To record read location.

(defvar charcolumn 0)        ; 0 means we have not printed a character yet.
(defvar charline 1)

(defvar charcolumn-linelength 80)

(defun bump-charcolumn (x &optional (n 0))
  (setq charcolumn (+ charcolumn (flatc x) n)))

(defun ll-terpri () (setq charcolumn 0) (incf charline))

(defun ll-tab (n)
  ;; This is tab-to.  After tabbing, the next character will
  ;; print in column n (0 based, i.e. 0 is the beginning of the line).
  (cond ((eql n charcolumn))
        ((> n charcolumn)
         (sloop for i from 1 to (- n charcolumn)
                do  (princ '#\space)))
        (t (ll-terpri) (ll-tab n)))
  (setq charcolumn n))

(defun ll-princ (x)  (princ x) (bump-charcolumn x))

(defmacro peekc () `(peek-char nil *standard-input* nil eof-value))

(defvar last-character-read nil)

(defmacro tyi ()
  '(progn (setq last-character-read
                (read-char *standard-input* nil eof-value))
          (if echo (write-char last-character-read *terminal-io*))
          (incf charcolumn)
          (when (newlinep last-character-read)
                (setq charcolumn 0)
                (setq charline (+ charline 1)))
          last-character-read))

;; utility

(defmacro if-echo (&rest x)
    `(if echo (let ((*standard-output* *terminal-io*))
                (progn . ,x))))

(defun string-append (a &rest b)
  (string-append2 a b))

(defun stringify (a)
  (cond ((null a) "")
        ((consp a)
         (concatenate 'string (stringify (car a)) (stringify (cdr a))))
        (t (string a))))

(defun string-append2 (a b)
  (if (null b)
      (stringify a)
      (concatenate 'string (stringify a)
                   (string-append2 (car b) (cdr b)))))

(defun pack2 (a b) (intern (string-append a b)))

(defun flatc (x) (length (princ-to-string x)))

(defun flatsize (x) (length (prin1-to-string x)))

(defun explode (x)
  (setq x (string x))
  (sloop for i from 0 to (1- (flatc x))
        collecting (char x i)))

(defun explodec (x)
  (setq x (string x))
  (sloop for i from 0 to (1- (flatc x))
        collecting (char-code (char x i))))
)

;;  Parser utilities

(proclaim '(special ll=line-errors token token-type token-loc
                    previous-token previous-token-type previous-token-loc))

(defun write-errors ()
  ;; This was derived from code that worked for semantic errors also.
  ;; This function only applies to syntactic errors.
  ;; ll=line-errors is of the form
  ;;   ( ((line-of-token column-of-token) . error-msg-string ) ... )
  ;; Errors will appear as:
  ;;  foo := from + or x;
  ;;            ^1     ^2
  ;;      1: Expected identifier, parens, ...
  ;;      2: Expected <expression> ....
  ;; 18 October 1983 by MKSmith
  (let ((pos 0) nextpos (num 0))
    ;; First print carats under positions at which tokens in errors begin.
    ;; NUM indicates the correspondence between token and msg.
    (mapc (function (lambda (err)
               (setq nextpos (cadr (car err)))
               (cond ((< pos nextpos)
                      (ll-tab nextpos)
                      (setq pos nextpos)
                      (ll-princ '^)
                      (ll-princ (incf num))))))
          ll=line-errors)
    ;; NUM indicates the correspondence between token and msg.
    (setq num 0)
    (mapc (function (lambda (err) (format t "    ~a: ~a" (incf num) (cdr err))))
          ll=line-errors)
    (setq ll=line-errors nil)))

(defun msg-print (n msg)
  ;;  Print n: <elements of MSG seperated by spaces>
  (let (max)
    (setq max charcolumn-linelength)
    (format t "~%    ~a : " n)
    (mapc (function (lambda (entry)
               (if (< (+ charcolumn (flatc entry)) max)
                   (format t " ~a" entry)
                   (format t "~%         ~a" entry))
               (bump-charcolumn entry 1)))
          msg)))

(defun ada-error (string)
  (format t "~%Error in interpreter: ~A~%" string)
  nil)

(defun ps1-error (x)
  ;; Modified to interact with the array form of Ada linereading.
  ;; X should be a string equal to the error message.
  (declare (special prs\err))
  (setq prs\err t)
  (setq ll=line-errors (nconc1 ll=line-errors (cons token-loc x))))

(defun ps1-warning (x)
  ;; Modified to interact with the array form of Ada linereading. X
  ;;  should be a list equal to the error message. We do not set
  ;;  PRS\ERR, so hopefully this will print as a warning, but not
  ;;  destroy the syntactic parse.
  (setq ll=line-errors (nconc1 ll=line-errors (cons token-loc x))))
```

References

[DoD 83]	*Reference Manual for the Ada Programming Language* United States Department of Defense, New York, 1983. ANSI/MIL-STD-1815A-1983.
[Steele 84]	Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

Index

Table of Contents

List of Figures

List of Tables