SRC Technical Note

1997-005c

June 2, 1998

The Vesta-2 Software Description Language

Allan Heydon, Jim Horning, Roy Levin, Timothy Mann, and Yuan Yu

Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://www.research.digital.com/SRC/

Table of Contents

1. Introduction

This note describes the formal syntax and semantics of the Vesta-2 Software Description Language (SDL). We expect it will be used as a reference by Vesta-2 users. Although the description is meant to be complete and unambiguous, it is by no means a language tutorial or user guide.

Vesta-2 is a software configuration management system [1]. Developers use Vesta-2 to build and manage potentially large-scale software. In Vesta-2, the instructions for building a software artifact are written as an SDL program. Evaluating the program causes the software system to be constructed; the program's result value typically contains the derived files produced by the evaluation.

Vesta-1, the precursor of Vesta-2, saw extensive use at the Digital Systems Research Center [2, 3, 4, 5]. Vesta-2 adopts many of the same concepts as Vesta-1, but Vesta-2 features substantial design changes (including major changes to the syntax and semantics of the SDL itself) and a portable implementation. In the rest of this note, references to ``Vesta" mean ``Vesta-2".

The Vesta SDL is a functional language with lexical scoping. Its value space includes Booleans, integers, texts, lists (similar to LISP lists), sequences of name-value pairs called *bindings*, closures, and a unique error value.

The language is dynamically typed; that is, types are associated with run-time values instead of with static names and expressions. Even without static type checking, the language is strongly typed: an executing Vesta program cannot breach the language's type system. The expected types of parameters to language primitives are defined, and those types are checked when the primitives are evaluated. The language includes provisions for specifying the types of used-defined function arguments and local variables, but these type declarations are currently unchecked.

The language contains roughly 60 primitive functions. There is a single `_run_tool` primitive for invoking external tools like compilers and linkers as function calls. External tools can be invoked from Vesta without modification.

Conceptually, every software artifact built with Vesta is constructed from scratch, thereby guaranteeing that the resulting artifact is composed of consistent pieces. Vesta uses extensive caching to avoid unnecessary rebuilding. Vesta records software dependencies automatically. The techniques by which the implementation caches function calls and determines dependencies are described in the complete Vesta-2 paper [1].

2. Lexical Conventions

The language semantics presented in Section 3 introduces each language construct by giving its syntax and semantics. This section defines the meta-notation and terminals assumed by the presented syntax fragments. The complete language syntax is given in Section 4.

2.1 Meta-notation

Nonterminals of the grammar begin with an uppercase letter, are at least two characters in length, and include at least one lowercase letter. Except for the four terminals listed in Section 2.2 below, each of which denotes a class of tokens, the terminals of the grammar are character strings not of this form.

The grammar is written in a variant of BNF (Backus-Naur Form). The meta-characters of this notation are:

```
::=  |  [  ]  {  }  *  +  `  '
```

The meaning of the metacharacters is as follows:

```
NT ::= Ex  NT rewrites to Ex
Ex1 | Ex2  Ex1 or Ex2
[ Ex ]     optional Ex
{ Ex }     meta-parentheses for grouping.
Ex*        zero or more Ex's
Ex*,       zero or more Ex's separated by commas, trailing comma optional
Ex*;       zero or more Ex's separated by semicolons, trailing optional
Ex+        one or more Ex's
Ex+,       one or more Ex's separated by commas, trailing comma optional
Ex+;       one or more Ex's separated by semicolons, trailing optional
`s'        the literal character or character sequence s
```

When used as terminals, square brackets, curly brackets, and vertical bar appear in single quotes to avoid ambiguity with the corresponding metacharacters (i.e., `` `[' ``, `` `]' ``, `` `{' ``, `` `}' ``, `` `|' ``).

2.2 Terminals

The following names are used as terminals in the grammar. They denote classes of tokens, and are defined precisely in Section 4.3

`Delim`
: A pathname delimiter. Either forward or backward slashes are allowed within pathnames, but not both.

`Integer`

An integer, expressed in either decimal, octal, or hexadecimal.

`Id`

An identifier. An identifier is any sequence of letters, digits, periods, and underscores that does not represent an integer. For example, foo and 36.foo are identifiers, but 36 and 0x36 are not.

`Text`

A text string. Texts are enclosed in double-quotes. They may contain escape sequences and spaces.

Comments and white space follow C++ conventions. A comment either begins with // and ends with the first subsequent newline or begins with /* and ends with */ (the latter form does not nest). Of course, these delimiters are only recognized outside text literals. White space delimits tokens but is otherwise ignored (except that the Space character, the ASCII character represented by the decimal number 32, is significant within text literals). The grammar prohibits white space other than the Space character within text literals.

The names of the built-in functions begin with an underscore character, and the identifier consisting of the single character "." plays a special role in the Vesta SDL. It is therefore recommended that Vesta programs avoid defining identifiers of these forms.

3. Semantics

The semantics of programs written in the Vesta SDL are described by a function *Eval* that maps a syntactic *expression* and a *context* to a *value*. That is, Eval(E, C) returns the value of the syntactic expression E in the context C. In addition to syntactic expressions (denoted by the non-terminal Expr in the grammar), the domain of Eval includes additional syntactic constructs. Some of these additional constructs are defined by the concrete grammar, while others are introduced as ``intermediate results" during the evaluation process (the latter are noted where they are introduced). Each value returned by Eval is in the Vesta value space, described in the next section. The context parameter C to Eval is a value of type t_binding in the Vesta value space.

3.1 Value Space

Values are typed. The types and values of the language are:

```
Type name     Values of the type
-------------------------------------------------------------------------
t_bool        true, false
t_int         integers
t_text        arbitrary byte sequences
t_list        sequences of zero or more arbitrary values
t_binding     sequences of zero or more pairs, in which the first member
                of each pair is a non-empty t_text, the second is an
                arbitrary value, and the first members of all the pairs
                are distinct
t_closure     closures, each of which is a triple <e, f, b> where
                e is a function body (i.e., a Block as per the grammar),
                f is a list of pairs <t_i, e_i>, where t_i is a
                  t_text value (a formal parameter name) and e_i is either
                  the distinguished expression <emptyExpr> or is
                  an Expr (for a default parameter value)
                b is a value of type t_binding (the context)
t_err         err
```

The values *true*, *false*, *emptylist* (the list of length zero), *emptybinding* (the binding of length zero), and *err* are

not to be confused with the language literals `TRUE`, `FALSE`, `<>`, `[]`, and `ERR` that denote those values.

The following supertype is used chiefly for defining the domain of primitive functions (the U(...) notation is type union):

```
t_value       U(t_bool, t_int, t_text, t_list,
                t_binding, t_closure, t_err)
```

The type t_bool contains the Boolean values *true* and *false*, denoted in the language by the literals `TRUE` and `FALSE`.

The type t_int contains integers over at least the range -2^31 .. 2^31-1; the exact range is implementation dependent.

The type t_text contains arbitrary sequences of 8-bit bytes. This type is used to represent text literals (quoted strings) in SDL programs as well as the contents of files introduced through the Files nonterminal of the grammar. Consequently, an implementation must reasonably support the representation of large values of this type (thousands of bytes or more), but is not required to support efficient operations on large text values.

The type t_list contains sequences of values. The elements of a list need not be of the same type.

The type t_binding contains sequences of pairs <*t_i*, *v_i*>, in which each *t_i* is a non-empty value of type t_text, each *v_i* is an arbitrary Vesta value (i.e., of type t_value), and the *t_i* are all distinct. Note that bindings are sequences: they are ordered. The *domain* of a binding is the set of names *t_i* at its top level. Bindings may be nested.

Bindings play an important role in the Vesta language. They are used to represent a variety of interesting objects. For example, flat bindings that map names to texts can be used to represent command-line switches and environment variables; bindings that contain nested bindings can be used to represent file systems; and bindings that map names to closures can be used to represent interfaces. Section 3.4.5 describes the primitive functions and operators for manipulating bindings, including three primitives for combining two bindings.

The type t_closure contains closure values for the primitive operators and functions (defined in Section 3.4) as well as for user-defined functions.

The type t_err consists of the single distinguished value *err*, denoted in the language by the literal `ERR`, which is used to represent erroneous evaluations. Primitive functions return *err* when applied to values outside their natural domain. For most (but not all) primitives, the value *err* lies outside the natural domain and so is ``contagious''; that is, most primitives return *err* when given *err* for any input. The evaluation rules and the descriptions of primitive functions document these cases.

In most cases, *err* represents a definite error and the implementation should generate a suitable diagnostic for human consumption, in addition to merely propagating the *err* value through subsequent evaluation. Whether the evaluation terminates or continues in these cases is left to the implementation.

3.2 Type Declarations

The language includes a rudimentary mechanism for declaring the expected types of values computed during

evaluation. The grammar defines a small sub-language of type expressions, which includes the ability to give names to types and to describe aggregate types (lists, bindings, functions) with varying degrees of detail. Type expressions may be attached to function arguments and results and to local variables, indicating the type of the expected value for these identifiers and expressions during evaluation.

The Vesta evaluator currently treats type names and type expressions as syntactically checked comments; it performs no other checking. Future implementations may type-check expressions at run-time and report an error if the value does not match the specified type (according to some as yet unspecified definition of what it means for a value to ``match" a type specification).

The syntax fragments and semantic descriptions in subsequent sections omit any further reference to type expressions entirely.

3.3 Evaluation Rules

The evaluation of a Vesta program corresponds to the abstract evaluation:

```
Eval( M([]) , C_initial)
```

where *M* is the closure corresponding to the contents of an immutable file (a system model) in the Vesta repository and *C_initial* is an initial context. *M*'s model should have the syntactic form defined by the nonterminal Model described in Section 3.3.13 below. *C_initial* defines the names and associated values of the built-in primitive operators and functions described in Section 3.4 below.

The definition of Eval by cases follows. Unless E is handled by one of these cases, Eval(E, C) is *err*. As mentioned above, the domain of Eval includes the language generated by the concrete grammar as a proper subset. Thus, in some of the cases below, the expression E can arise only as an intermediate result of another case of Eval. These cases are explicitly noted.

The pseudo-code that defines the various cases of Eval and the primitive functions should be read like C++. That code assumes the following declaration for the representation of Vesta values:

```
class val {
  public:
    operator int();
    // converts Vesta t_int or t_bool to C++ int

    val(int);
    // converts a C++ integer to a Vesta t_int

    int operator== (val);
    // compares two Vesta values, returning true (1)
    // if they have the same type and are equal, and
    // false (0) otherwise
}
```

Note that the `operator==` above is the one invoked by uses of ``==" in the C++ pseudo-code. It is not to be confused with the primitive equality operator defined on various Vesta types in Section 3.4.

The pseudo-code also refers to the following constants:

```
  static val true;         // value of literal TRUE
  static val false;        // value of literal FALSE
  static val emptylist;    // value of literal < >
  static val emptybinding; // value of literal [ ]
  static val err;          // value of literal ERR
```

For convenience, the pseudo-code adopts the following notational conveniences:

- Eval is defined by cases rather than by one C++ function with an enormous embedded case selection.
- Recursive references to Eval appear inline in the same form that is used to identify the individual cases.
- Primitive functions of the Vesta language, whose names begin with an underscore, are invoked inline from the pseudo-code as if they were ordinary C++ functions. The primitive operators of the Vesta language are invoked this way too; for example, when the pseudo-code refers to operator+, it means the Vesta primitive function, not the C++ operator. Note that some of the Vesta operators are overloaded by type, but not by arity. For example, operator+ is defined on integers, texts, lists, and bindings, but it always takes two arguments.
- In the pseudo-code for rules that contain the terminal Id, the variable `id` denotes the value of the Id represented as a t_text.

In each of the following sections, we first present the relevant portions of the language syntax. We then present the evaluation rules that apply to those syntactic constructs. The complete language syntax is given in Section 4.

3.3.1 Expr

Syntax:

```
Expr       ::= if Expr then Expr else Expr | Expr1
Expr1      ::= Expr2 {  =>  Expr2 }*
Expr2      ::= Expr3 {  ||  Expr3 }*
Expr3      ::= Expr4 {  &&  Expr4 }*
Expr4      ::= Expr5 [ { == | != | < | > | <= | >= } Expr5 ]
Expr5      ::= Expr6 { AddOp Expr6 }*
AddOp      ::= +  |  ++  |  -
Expr6      ::= Expr7 { MulOp Expr7 }*
MulOp      ::= *
Expr7      ::= [ UnaryOp ] Expr8
UnaryOp    ::= -  |  !
Expr8      ::= Primary [ TypeQual ]
Primary    ::= ( Expr ) | Literal | Id | List
             | Binding | Select | Block | FuncCall
```

The grammar lists the operators in increasing order of precedence. The binary operators at each precedence level are left-associative.

Evaluation Rules:

```
// conditional expression
Eval( if Expr_1 then Expr_2 else Expr_3 , C) =
```

```
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) return err;
  if (b == true) return Eval( Expr_2 , C);
  else return Eval( Expr_3 , C);
}
```

As defined in Section 3.4.6, `_is_bool(b)` is *true* if *b* is a value of type t_bool and *false* otherwise.

```
// conditional implication
Eval( Expr_1 => Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) return err;
  if (b == false) return true;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) return err;
  return b;
}

// conditional OR
Eval( Expr_1 || Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) return err;
  if (b == true) return true;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) return err;
  return b;
}

// conditional AND
Eval( Expr_1 && Expr_2 , C) =
{
  val b = Eval( Expr_1 , C);
  if (_is_bool(b) == false) return err;
  if (b == false) return false;
  b = Eval( Expr_2 , C);
  if (_is_bool(b) == false) return err;
  return b;
}

// comparison
Eval( Expr_1 == Expr_2 , C) = operator==(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 != Expr_2 , C) = operator!=(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 <  Expr_2 , C) = operator< (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 >  Expr_2 , C) = operator> (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 <= Expr_2 , C) = operator<=(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 >= Expr_2 , C) = operator>=(Eval( Expr_1 , C), Eval( Expr_2 , C))

// AddOp and MulOp
Eval( Expr_1 +  Expr_2 , C) = operator+ (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 ++ Expr_2 , C) = operator++(Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 -  Expr_2 , C) = operator- (Eval( Expr_1 , C), Eval( Expr_2 , C))
Eval( Expr_1 *  Expr_2 , C) = operator* (Eval( Expr_1 , C), Eval( Expr_2 , C))

// UnaryOp
```

```
Eval( ! Expr , C) = operator!(Eval( Expr , C))
Eval( - Expr , C) = operator-(Eval( Expr , C))

// parenthesization
Eval( ( Expr ) , C) = Eval( Expr , C)
```

There are seven remaining possibilities for a Primary: Literal, Id, List, Binding, Select, Block, and FuncCall. These are treated separately in subsequent sections.

3.3.2 Literal

Syntax:

```
Literal    ::= ERR | TRUE | FALSE | Text | Integer
```

Evaluation Rules:

```
Eval( ERR    , C) = err
Eval( TRUE   , C) = true
Eval( FALSE  , C) = false
Eval( Text   , C) = the corresponding t_text value, following the C++
                    interpretation for the Escape characters.
Eval( Integer, C) = the corresponding t_int value if it can be
                    represented by the implementation, otherwise `err'.
```

3.3.3 Id

Evaluation Rules:

```
Eval( Id , C) = _lookup(C, id),
```

As defined in Section 3.4.5, `_lookup(`*b*`, `*nm*`)` is the value associated with the non-empty name *nm* in the binding *b*, or *err* if *nm* is empty or is not in *b*'s domain.

3.3.4 List

Syntax:

```
List       ::= < Expr*, >
```

The use of <, > as both binary operators and list delimiters makes the grammar ambiguous. Section 4.2 explains how the ambiguity is resolved.

Syntactic desugarings:

```
< Expr_1, ..., Expr_n >  desugars to  < Expr_1 > + < Expr_2, ..., Expr_n >
```

Here, `+' is the concatenation operator on lists.

Evaluation Rules:

```
Eval( <>       , C) = emptylist
```

```
Eval( < Expr > , C) = _list1(Eval( Expr , C))
```

As defined in Section 3.4.4, `_list1(val)` evaluates to a list containing the single value *val*.

3.3.5 Binding

Syntax:

```
Binding    ::= `[' BindElem*, `]'
BindElem   ::= SelfNameB | NameBind
SelfNameB  ::= Id
NameBind   ::= GenPath = Expr
GenPath    ::= GenArc { Delim GenArc }* [ Delim ]
GenArc     ::= Arc | $ Id | $ ( Expr ) | % Expr %
Arc        ::= Id | Integer | Text
```

Syntactic desugarings:

The following desugarings apply to BindElem's within a Binding.

```
Id                          desugars to  Id = Id
GenArc Delim = Expr         desugars to  GenArc = Expr
GenArc Delim GenPath = Expr desugars to  GenArc = [ GenPath = Expr ]
$ Id = Expr                 desugars to  $ ( Id ) = Expr
% Expr_1 % = Expr_2         desugars to  $ ( Expr_1 ) = Expr_2
```

The SelfNameB syntactic sugar allows names from the current scope to be copied into bindings more succinctly. For example, the binding value:

```
  [ progs = progs, tests = tests, lib = lib ]
```

can instead be written:

```
  [ progs, tests, lib ]
```

The GenPath syntactic sugar allows bindings consisting of a single path to be written more succinctly. For example, the binding value:

```
  [ env_ovs = [ Cxx = [ switches = [ compile =
    [ debug = "-g3", optimize = "-O" ]]]]]
```

can instead be written:

```
  [ env_ovs/Cxx/switches/compile =
    [ debug = "-g3", optimize = "-O" ]]
```

Evaluation Rules:

First, the rules for constructing empty and singleton bindings:

```
Eval( [ ]            , C) = emptybinding
Eval( [ Arc = Expr ] , C) = _bind1(id, Eval( Expr , C))
```

Here *id* is the t_text representation of *Arc*. The conversion from an Arc to a t_text is straightforward. If the Arc is an Id, the literal characters of the identifier become the text value. If the Arc is an Integer, the literal characters used to represent the integer in the source of the model become the text value. If the Arc is a Text, the result of Eval(Arc, C) is used. As defined in Section 3.4.5, `_bind1(id, v)` evaluates to a singleton binding that associates the non-empty t_text *id* with the value *v*.

The `$(Expr)` syntax allows the name introduced into a binding to be computed:

```
Eval( [ $ ( Expr_1 ) = Expr_2 ] , C) =
  _bind1(Eval(Expr_1, C), Eval( Expr_2 , C))
```

When the field name is computed using the `$` syntax, an empty string is illegal (see _bind1 below), and the expression must evaluate to a t_text.

The following rule handles the case where multiple BindElem's are given.

```
Eval( [ BindElem_1, ..., BindElem_n ] , C) =
  _append(Eval( [ BindElem_1 ] , C),
          Eval( [ BindElem_2, ..., BindElem_n ] , C)
```

As defined in Section 3.4.5, `_append(b1, b2)` evaluates to the concatenation of the bindings *b1* and *b2*; it requires that their domains are disjoint.

3.3.6 Select

Syntax:

```
Select     ::= Primary Selector GenArc
Selector   ::= Delim | !
GenArc     ::= Arc | $ Id | $ ( Expr ) | % Expr %
Arc        ::= Id | Integer | Text
```

A Select expression denotes a selection from a binding, so the Primary must evaluate to a binding value.

Syntactic Desugarings:

```
Primary Selector % Expr %  desugars to  Primary Selector $ ( Expr )
```

Evaluation Rules:

The Delim syntax selects a value out of a binding by name.

```
Eval( Primary Delim Arc , C) =
  _lookup(Eval( Primary , C), id)
```

Here *id* is the t_text value of *Arc*, as defined in Section 3.3.5 above.

The `$(Expr)` syntax allows the selected name to be computed:

```
Eval( Primary Delim $ ( Expr ) , C) =
  _lookup(Eval( Primary , C), Eval( Expr , C))
```

The `!` syntax tests whether a name is in a binding's domain:

```
Eval( Primary ! Id , C) =
  _defined(Eval( Primary , C), id),
```

As defined in Section 3.4.5, `_defined(`*b*`, `*nm*`)` evaluates to *true* if *nm* is non-empty and in *b*'s domain, and to *false* otherwise.

As above, the `$(`*Expr*`)` syntax can be used to compute the name:

```
Eval( Primary ! $ ( Expr ) , C) =
  _defined(Eval( Primary , C), Eval( Expr , C))
```

In both cases where the GenArc is a computed expression, the Expr must evaluate to a t_text.

3.3.7 Block

Syntax:

```
Block      ::= `{' Stmt*; Result; `}'
Stmt       ::= Assign | Iterate | FuncDef | TypeDef
Result     ::= { value | return } Expr
```

Syntactic Desugarings:

```
return Expr   desugars to   value Expr
```

That is, the keywords `return` and `value` are synonyms, provided for stylistic reasons. The `return`/`value` statement must appear at the end of a Block; there is no analog of the C/C++ return statement that terminates execution of the function in which it appears.

Evaluation Rules:

Since the Vesta SDL is functional, evaluation of a statement does not produce side-effects, but rather produces a binding. Evaluation of a block occurs by augmenting the context with the bindings produced by evaluating the Stmts, then evaluating the final Expr in the augmented context.

```
Eval( { value Expr } , C) = Eval( Expr , C)

Eval( { Stmt_1; ...; Stmt_n; value Expr } , C) =
  Eval( Expr , operator+(C, Eval( { Stmt_1; ...; Stmt_n } , C)))
```

Notice that this second rule introduces an argument to Eval in the ``extended" language that is not generated by any non-terminal of the grammar.

3.3.8 Stmt

Evaluation Rules:

Evaluating a Stmt or sequence of Stmts produces a binding. Note that the binding resulting from the evaluation of

a sequence of Stmts is simply the overlay (operator `+') of the bindings resulting from evaluating each Stmt in the sequence, and does not include the context *C*.

```
Eval( { } , C) = emptybinding

Eval( { Stmt_1; Stmt_2 ...; Stmt_n } , C) =
{
  val b = Eval( Stmt_1 , C);
  return operator+(b, Eval( { Stmt_2; ...; Stmt_n } , operator+(C, b)))
}
```

These rules apply to constructs in the ``extended" language. There are three possibilities for a Stmt: Assign, Iterate, and FuncDef. They are covered in the next three sections.

3.3.9 Assign

Since the Vesta SDL is functional, assignments do not produce side-effects. Instead, they introduce a new name into the evaluation context whose value is that of the given expression.

Syntax:

```
Assign     ::= Id [ TypeQual ] [ Op ] = Expr
Op         ::= AddOp | MulOp
AddOp      ::= +  |  ++  |  -
MulOp      ::= *
```

Syntactic Desugarings:

```
Id Op = Expr   desugars to   Id = Id Op Expr
```

Evaluation Rules:

```
Eval( Id = Expr , C) = _bind1(id, Eval( Expr , C))
```

3.3.10 Iterate

The language includes expressions for iterating over both lists and bindings. There is also a `_map` primitive defined on lists (Section 3.4.4) and bindings (Section 3.4.5). `_map` is more efficient but less general than the language's Iterate construct.

Syntax:

```
Iterate    ::= foreach Control in Expr do IterBody
Control    ::= Id | `[' Id = Id `]'
IterBody   ::= Stmt | `{' Stmt+; `}'
```

The two Control forms are used to iterate over lists and bindings, respectively.

Evaluation Rules:

```
// iteration with single-statement body
Eval( foreach Control in Expr do Stmt , C) =
```

```
  Eval( foreach Control in Expr do { Stmt } , C)
```

The semantics of a loop are to conceptually unroll the loop *n* times, where *n* is the length of the list or binding being iterated over.

```
// iteration over a list
Eval( foreach Id in Expr do { Stmt_1; ...; Stmt_n } , C) =
{
  val l = Eval( Expr, C);
  if (_is_list(l) == false) return err;
  t_text id = Id; // identifier Id as a t_text
  val r = emptybinding;
  for (; !(l == emptylist); l = _tail(l)) {
    val r1 = operator+(C, r);
    r1 = operator+(r1, _bind1(id, _head(l)));
    r = operator+(r, Eval( { Stmt_1; ...; Stmt_n } , r1));
  }
  return r;
}
```

As defined in Section 3.4.6, `_is_list(l)` is *true* if *l* is of type t_list, and *false* otherwise.

```
// iteration over a binding
Eval( foreach [ Id1 = Id2 ] in Expr do { Stmt_1; ...; Stmt_n } , C) =
{
  val b = Eval( Expr, C);
  if (_is_binding(b) == false) return err;
  t_text id1 = Id1; // identifier Id1 as a t_text
  t_text id2 = Id2; // identifier Id2 as a t_text
  val r = emptybinding;
  for (; !(b == emptybinding); b = _tail(b)) {
    val r1 = operator+(C, r);
    r1 = operator+(r1, _bind1(id1, _n(_head(b))));
    r1 = operator+(r1, _bind1(id2, _v(_head(b))));
    r = operator+(r, Eval( { Stmt_1; ...; Stmt_n } , r1));
  }
  return r;
}
```

As defined in Section 3.4.6, `_is_binding(b)` is *true* if *b* is of type t_binding, and *false* otherwise.

Note that the iteration variables (that is, `Id`, `Id1`, and `Id2` above) are not bound in the binding that results from evaluating the `foreach` statement. However, any assignments made in the loop body *are* included in the result binding.

Iteration statements are typically used to walk over or collect parts of a list or binding. For example, here is a function for reversing a list:

```
  reverse_list(l: list): list
  {
    res: list = <>;
    foreach elt in l do
      res = <elt> + res;
    return res;
```

```
  }
```

Here is a function that counts the number of leaves of a binding:

```
  count_leaves(b: binding): int
  {
    res: int = 0;
    foreach [ nm = val ] in b do
      res += if _is_binding(val) then count_leaves(val) else 1;
    return res;
  }
```

3.3.11 FuncDef

Syntax:

The function definition syntax allows a suffix of the formal parameters to have associated default values.

```
FuncDef    ::= Id Formals+ [ TypeQual ] Block
Formals    ::= ( FormalArgs )
FormalArgs ::= { TypedId*,                                    // none defaulted
             | { TypedId = Expr }*,                           // all defaulted
             | TypedId { , TypedId }* { , TypedId = Expr }+ } // some defaulted
```

Note that the syntax allows multiple `Formals` to follow the function name. As the rules below describe, the use of multiple Formals produces a sequence of curried functions, all but the first of which is anonymous.

Evaluation Rules:

```
Eval( Id Formals_1 ... Formals_n Block , C) =
  _bind1(id, Eval( e , C1)),
  where:
    e = LAMBDA Formals_1 ... LAMBDA Formals_n Block
    C1 = operator+(C, _bind1(id, Eval( e , C1)))
```

Notice the recursive definition of C1. This permits functions to be self-recursive, but not mutually recursive. Although this recursive definition looks a little odd, it can be implemented by the evaluator by introducing a cycle into the context C1. This is the only case where any Vesta value can contain a cycle (the language syntax and operators do not allow cyclic lists or bindings to be constructed), and the cycle is invisible to clients. There is no practical difficulty in constructing the cycle because, as we are about to see, the ``evaluation" of a `LAMBDA` is purely syntactic.

Also note that this rule produces a `LAMBDA` construct in the ``extended" language that is not generated by any non-terminal of the grammar. The following is the simple case of `LAMBDA`, where all actual parameters must be given in any application of the closure. The reason for the restriction on the use of "." as a formal parameter is treated below in the section on function calls.

```
Eval( LAMBDA (Id_1, ..., Id_m)
      LAMBDA Formals_2 ... LAMBDA Formals_n Block , C) =
  If any of the Id's is the identifier ".", return err; otherwise,
  return the t_closure value
    <LAMBDA Formals_2 ... LAMBDA Formals_n Block, f, C>, where:
```

```
      f is a list of pairs <id_i, <emptyExpr>> where:
        id_i is the t_text representation of Id_i, for i in [1..m]
```

In the typical case where only one set of Formals is specified (that is, *n* = 1), the first element of the resulting closure value is simply a Block.

Next is the general case of LAMBDA, in which ``default expressions'' are given for a suffix of the formal parameter list. Functions may be called with fewer actuals than formals if each formal corresponding to an omitted actual includes an expression specifying the default value to be computed. When the closure is applied, if an actual parameter is missing, its formal's expression is evaluated (in the context of the LAMBDA) and passed instead. The following FuncCall section defines this precisely.

```
Eval( LAMBDA (Id_1, ..., Id_k, Id_k+1 = Expr_k+1, ... Id_m = Expr_m)
      LAMBDA Formals_2 ... LAMBDA Formals_n Block , C) =
  If any of the Id's is the identifier ".", return err; otherwise,
  return the t_closure value
    <LAMBDA Formals_2 ... LAMBDA Formals_n Block, f, C>, where:
       f is a list of pairs <id_i, expr_i> where:
         id_i is the t_text representation of Id_i, for i in [1..m]
         expr_i is <emptyExpr>, for i in [1..k],
         expr_i is Expr_i, for i in [k+1..m]
```

3.3.12 FuncCall

Syntax:

```
FuncCall   ::= Primary Actuals
Actuals    ::= ( Expr*, )
```

Evaluation Rules:

The function call mechanism provides special treatment for the identifier consisting of a single period, called the *current environment* and pronounced ``dot''. Dot is typically assigned a binding that contains the tools, switches, and file system required for the rest of the build. The initial environment, *C_initial*, does not bind dot (that is, ```_defined(C_initial, ".") == false`''').

When a function is called, the context in which its body executes may bind "." to a value established as follows:

- if the function is defined with *n* formals and called with *n* or fewer actuals, then the value for "." at the point of call is bound to the implicit formal parameter named "." in the callee;
- if the function is defined with *n* formals and called with *n*+1 actuals, then the value bound to the implicit formal parameter named "." is the value of the last actual.

Thus, the binding for ".", if any, is passed through the dynamic call chain until it is altered either explicitly by an Assign statement or implicitly by calling a function with an extra actual parameter. The pseudo-code below makes this precise.

```
Eval( Primary ( Expr_1, ..., Expr_n ) , C) =
{
  val cl = Eval( Primary , C);
```

```
  if (_is_closure(cl) == false) return err;

  // cl.e is the function body, cl.f are the formals, cl.b is the context
  int m = _length(cl.f);              // number of formals
  if (n > m + 1) return err;          // too many actuals
  val C1 = cl.b;                      // t_binding
  val f = cl.f;                       // t_list (of <t, e> pairs)

  // augment C1 to include formals bound to corresponding actuals
  int i;
  for (i = 1; i <= m; i++) {
    val form = _head(f);              // i-th formal (a <t, e> pair)
    val act;                          // value of corresponding actual
    if (i <= n)
      act = Eval( Expr_i , C);        // value for i-th actual
    else {
      if (form.e == <emptyExpr>)
        return err;                   // missing required actual
      act = Eval( form.e , cl.b);     // value for defaulted argument
    }
    C1 = operator+(C1, _bind1(form.t, act));
    f = _tail(f);
  }

  // bind "." in C1
  val dot;
  if (n <= m)
    dot = _lookup(C, ".");            // inherit value for "." from C
  else
    dot = Eval( Expr_n , C);          // explicit value for last actual
  C1 = operator+(C1, _bind1(".", dot));

  /* C1 is now a suitable environment.  If the closure is a primitive
     function, then invoke it by a special mechanism internal to the
     evaluator and return the value it computes.  Otherwise, perform
     the following: */
  return Eval( cl.e , C1);
}
```

Note: The comparison with <emptyExpr> has not been formalized, but it should be intuitively clear.

3.3.13 Model

Syntax:

```
Model      ::= Files Imports Block
```

Evaluation Rules:

The nonterminal Model is treated like the body of a function definition (i.e., like a FuncDef, but without the identifier naming the function and with an empty list of formal parameters). More precisely:

```
Eval( Files Imports Block , C) =
  Eval( LAMBDA () Block , _append(Eval( Files Imports , emptybinding), C))
```

As this rule indicates, the Files and Imports constructs are evaluated in an empty context, and they augment the closure context in which the model's `LAMBDA` is evaluated. In practice, the context C will always be the initial context C_initial when this rule is applied (cf. Sections 3.3 and 3.3.15).

The Files nonterminal introduces values corresponding to the contents of ordinary files and directories. The Imports nonterminal introduces closure values corresponding to other Vesta SDL models.

The evaluation rules handle Files and Imports clauses by augmenting the context using the _append primitive, thereby ensuring that the names introduced by these clauses are all distinct, just as if the Files and Imports clauses of the Model were a single binding constructor. The Files and Imports clauses are evaluated independently:

```
Eval( Files Imports , C) =
  _append(Eval( Files , C), Eval( Imports , C))
```

The following two sections give the rules for evaluating Files and Imports clauses individually. It is worth noting that the evaluation context C is ignored in those rules.

3.3.14 Files

A Files clause introduces names corresponding to files or directories in the Vesta repository. Generally, these files or directories are named by relative paths, which are interpreted relative to the location of the model containing the Files clause. Absolute paths are permitted, though they are expected to be rarely used.

Syntax:

```
Files       ::= FileClause*
FileClause  ::= files FileItem*;
FileItem    ::= FileSpec | FileBinding
FileSpec    ::= [ Arc = ] DelimPath
FileBinding ::= Arc = `[' FileSpec*, `]'

DelimPath   ::= [ Delim ] Path [ Delim ]
Path        ::= Arc { Delim Arc }*
Arc         ::= Id | Integer | Text
```

Each FileItem in a Files clause takes one of two forms: a FileSpec or a FileBinding. Each form introduces (binds) exactly one name. In the former case, the name corresponds to the contents of a single file or directory; in the latter case, the name corresponds to a binding consisting of perhaps many files or directories. In both cases, the identifier introduced into the Vesta naming context or the identifiers introduced into the binding can be specified explicitly or derived from an Arc in the Path.

For example, consider the following `files` clause:

```
  files
    scripts = bin;
    c_files = [ utils.c, main.c ];
```

Suppose the directory containing this model also contains a directory named `bin` and files named `utils.c` and `main.c`. Then this `files` clause introduces the two names `scripts` and `c_files` into the context. The former is bound to a binding whose structure corresponds to the `bin` directory. The latter is bound to a binding that maps the names `utils.c` and `main.c` to the contents of those files, respectively. The file contents are values of

type t_text.

Syntactic Desugaring:

When multiple FileItem's are given in a FileClause, the `files` keyword simply distributes over each of the FileItem's. That is:

```
  files FileItem_1; ...; FileItem_n;
```

desugars to:

```
  files FileItem_1;
  ...;
  files FileItem_n;
```

When the initial Arc is omitted from a FileSpec, it is inferred from the path. In particular:

```
  files [ Delim ] { Arc Delim }* Arc [ Delim ];
```

desugars to:

```
  files Arc = [ Delim ] { Arc Delim }* Arc [ Delim ];
```

Evaluation Rules:

Multiple FileClause's are evaluated independently:

```
Eval( FileClause_0 FileClause_1 ... FileClause_n , C) =
  _append(Eval( FileClause_0 , C), Eval( FileClause_1 ... FileClause_n , C))
```

That leaves only two cases to consider: FileSpec (in which the initial Arc is specified) and FileBinding.

```
// FileSpec
Eval( files Arc = DelimPath , C) = _bind1(id, v)
```

where:

- *id* is the t_text representation of *Arc*, as defined in Section 3.3.5 above.
- If *DelimPath* begins with a Delim, it is interpreted as an absolute path, which must nevertheless resolve to a file or directory in the Vesta repository. If *DelimPath* does not begin with a Delim, it refers to a file or directory named relative to the directory of the enclosing Model.
- If the entity named by *DelimPath* is a file, *v* is a t_text value formed by taking the file's contents. If *DelimPath* names a directory, *v* is a t_binding value constructed from the contents of the the directory, treating the files (if any) in the directory as above (i.e., as t_text values) and the directories (if any) recursively (i.e., as bindings). The members of the resulting binding are in an unspecified order. If *DelimPath* does not correspond to either an extant file or directory, *v* is the value *err*.

```
// FileBinding
Eval( files Arc = [ FileSpec_1, ..., FileSpec_n ] , C) =
```

```
  _bind1(id, Eval( files FileSpec_1; ...; FileSpec_n , C))
```

Again, *id* is the t_text representation of *Arc*.

The FileBinding form of the Files clause provides a convenient way to create a binding containing multiple FileSpecs. Without this construct, it would be necessary to name each file twice, once in the FileSpec and once in a subsequent binding constructor. Making a binding with FileBinding is semantically similar to constructing a file system directory, with the additional property that there is an enumeration order for the component files.

Notice that the grammar and evaluation rules given above for *FileSpec* and *FileBinding* allow a general Arc on the left-hand side of each equal sign, not just an Id. This was done to simplify the definitions and desugaring rules. However, it would be useless to write constructs like the following, which introduce names that cannot be referenced in the body of the model:

```
files
    33;
    34 = 34;
    "hash-table.c";
    "foo bar" = [ foo, bar ];
```

Therefore, we introduce an additional restriction: the context created by a Files clause must bind only names that are legal identifiers; that is, names that match the syntax of the Id token.

If you need to use files whose names are not legal identifiers, you should either assign them legal names with the equal sign syntax or embed them in a binding. Some possibilities:

```
// Choose a legal name
files
    f33 = 33;
    f34 = 34;
    hash_table.c = "hash-table.c";
    foo_bar = [ foo, bar ];

// Embed in a binding
files
    f = [ 33, 34 ];
    src = [ "hash-table.c" ];
```

3.3.15 Imports

The Imports clause enables one Vesta SDL model to reference and use others; that is, it supports modular decomposition of Vesta SDL programs.

Syntax:

```
Imports   ::= ImpClause*
ImpClause  ::= ImpIdReq | ImpIdOpt
```

There are two major forms of the Imports clause: one where identifiers are required (ImpIdReq), and one where they are optional (ImpIdOpt). Both forms have two sub-forms in which either a single model or a list of models may be imported.

First, consider the ImpIdReq case. This form is typically used to import models in the same package as the importing model. Each ImpItemR in the ImpIdReq clause takes one of two forms: an ImpSpecR or an ImpListR. Each form binds exactly one name.

```
ImpIdReq   ::= import ImpItemR*;
ImpItemR   ::= ImpSpecR | ImpListR
ImpSpecR   ::= Arc = DelimPath
ImpListR   ::= Arc = `[' ImpSpecR*, `]'

DelimPath  ::= [ Delim ] Path [ Delim ]
Path       ::= Arc { Delim Arc }*
Arc        ::= Id | Integer | Text
```

In the ImpSpecR case, the name is bound to the t_closure value that results from evaluation of the contents of a file according to the Model evaluation rules of Section 3.3.13. For example, consider the Import clause:

```
  import self = progs.ves;
```

This clause binds the name `self` to the closure corresponding to the local `progs.ves` model in the same directory as the model in which it appears.

In the ImpList case, the name is bound to a binding of such values. For example:

```
  import sub =
    [ progs = src/progs.ves, tests = src/tests.ves ];
```

This clause binds the name `sub` to a binding containing the names `progs` and `tests`; these names within the binding are bound to the closures corresponding to the models named `progs.ves` and `tests.ves` in the package's `src` subdirectory. For example, the `progs.ves` model would be invoked by the expression ``sub/progs()''.

Because the Imports clause often mentions several files with names that share a common prefix, a syntactic form is provided to allow the prefix to be written once. This is the ImpIdOpt form. It is used to import models from other packages. The semantics are defined so that many identifiers are optional; when omitted, they default to the name of the package from which the model is being imported. As in the ImpIdReq case, ImpIdOpt has forms for importing both single models and lists of multiple models.

```
ImpIdOpt   ::= from DelimPath import ImpItemO*;
ImpItemO   ::= ImpSpecO | ImpListO
ImpSpecO   ::= [ Arc = ] Path [ Delim ]
ImpListO   ::= Arc = `[' ImpSpecO*, `]'
```

Here are some examples of ImpIdOpt imports:

```
  from /vesta/src.dec.com/vesta import
    cache/12/build.ves;
    libs = [ srpc/2/build.ves, basics/5/build.ves ];
```

This example binds the name `cache` to the closure corresponding to version 12 of that package's `build.ves` model, and it binds the name `libs` to a binding containing the names `srpc` and `basics`, bound to versions 2 and 5 of those package's `build.ves` models. (As the evaluation rules below describe, the three occurrences of ``/build.ves'' in this example could actually have been omitted.)

Syntactic Desugaring:

When multiple ImpItemR's are given in a ImpIdReq, the `import` keyword distributes over each of the ImpItemR's. That is:

```
import ImpSpec_1; ...; ImpSpec_n;
```

desugars to:

```
import ImpSpec_1;
...;
import ImpSpec_n;
```

Similarly, the `from` clause distributes over the individual imports of an ImpIdOpt. In particular:

```
from DelimPath import ImpItemO_1; ...; ImpItemO_n;
```

desugars to:

```
from DelimPath import ImpItemO_1;
...;
from DelimPath import ImpItemO_n;
```

The use of `from` makes it optional to supply a name for the closure value being introduced; if the name is omitted, it is derived from the Path following the `import` keyword as follows:

```
from DelimPath import
  [ Arc_1 = ] [ Delim ] Arc_2 { Delim Arc }* [ Delim ]
```

desugars to:

```
import Arc =
  DelimPath Delim Arc_2 { Delim Arc }* [ Delim ]
```

where *Arc* is *Arc_1* if it is present and is *Arc_2* otherwise.

Similarly:

```
from DelimPath import Arc = [
  [ Arc1_1 = ] [ Delim ] Arc2_1 { Delim Arc }* [ Delim ],
  ...,
  [ Arc1_n = ] [ Delim ] Arc2_n { Delim Arc }* [ Delim ] ]
```

desugars to:

```
import Arc = [
  Arc_1 = DelimPath Delim Arc2_1 {Delim Arc }* [ Delim ],
  ...,
  Arc_n = DelimPath Delim Arc2_n {Delim Arc }* [ Delim ] ]
```

where *Arc_i* is *Arc1_i* if it is present and is *Arc2_i* otherwise.

Evaluation Rules:

Multiple ImpClause's are evaluated independently:

```
Eval( ImpClause_0 ImpClause_1 ... ImpClause_n , C) =
  _append(Eval( ImpClause_0 , C), Eval( ImpClause_1 ... ImpClause_n , C))
```

This leaves two fundamental forms of the Imports clause, whose semantics are defined as follows:

```
// ImpSpecR
Eval( import Arc = DelimPath , C) =
  _bind1(id, Eval( model , C_initial))
```

where:

- *id* is the t_text representation of *Arc*, as defined in Section 3.3.5 above.

- Let *f* be the sequence of Delims and Arcs that constitute the DelimPath.
    1. If *f* does not begin with a Delim, prepend ``Delim Path0 Delim'' to *f*, where *Path0* names the directory containing the Model in which this Imports clause appears.
    2. Lookup the path *f* in the Vesta repository. (See Filename Interpretation below.) If *f* names a directory, append a Delim (if *f* doesn't already end in one) and the string "build.ves", then lookup the augmented path *f* in the repository again. If *f* does not name a directory and its final element does not end in ".ves", append the string ".ves" to the final element of *f*, and look it up in the repository again.

- *model* is the Vesta SDL Model represented by the contents of the file in the Vesta repository named by the sequence *f*. If no such expression can be produced (e.g., the file doesn't exist, or can't be parsed as an expression), *model* is the expression ERR.

```
// ImpListR
Eval( import Arc = [ ImpSpecR_1, ..., ImpSpecR_n ] , C) =
  _bind1(id, Eval( import ImpSpecR_1; ...; ImpSpecR_n , C))
```

Again, *id* is the t_text representation of *Arc*.

As with the Files clause, and for the same reason, we add one restriction to the rules just given: the context created by an Imports clause must bind only names that are legal identifiers; that is, names that match the syntax of the Id token.

3.3.16 Filename Interpretation

The evaluation rules for the Files and Imports clauses do not specify how the sequence of Arcs and Delims making up a DelimPath is converted into a filename in the underlying file system. While this is somewhat system-dependent, it is nevertheless intended to be intuitive. In particular,

- Multiple adjacent Delims are replaced by a single one. (The grammar above doesn't permit adjacent Delims, but they can be produced by the desugaring rules.)

- The Vesta SDL syntax allows the arbitrary intermingling of ``/" and ``\" as arc separators. However, the implementation actually requires that Vesta programs use one or the other uniformly. When creating a filename from a sequence of Arcs and Delims, the implementation inserts the appropriate arc separator required by the underlying file system. The choice is not influenced by the choice of Delim that appears in the Vesta SDL program.

- The grammar permits an Arc to be an arbitrary Text. An Arc in a filename, however, is forbidden to contain a Delim character (i.e., forward or backward slash), and the Arcs ``. ." and ``." are forbidden in filenames as well. In particular, ``. ." cannot be used to mean *parent directory* and ``." cannot be used to mean *current directory*. The ``. ." notation is forbidden for technical reasons related to Vesta caching, while the ``." notation is simply unimplemented. However, the empty Arc ``" can be used to denote the current directory.

3.4 Primitives

The primitive names and associated values described below are provided by the Vesta SDL interpreter in *C_initial*, the initial context. Most of these values are closures with empty contexts; that is, they are primitive functions.

In the descriptions that follow, the notation used for the function signatures follows C++, with the result type preceding the function name and each argument type preceding the corresponding argument name. Defaulting conventions also follow C++; if an argument name is followed by "`= <value>`", then omitting the corresponding actual argument is equivalent to supplying <value>.

Some of the function signatures use the C++ operator definition syntax, which should be understood as defining a function whose name is not an Id in the sense of the grammar above. Such operator names cannot be rebound. These operators are typically overloaded, as the descriptions below indicate. Uses of these built-in Vesta primitives within C++ code are denoted by the `operator` syntax.

The pseudo-code of this section assumes the definition of the Vesta value class given at the start of Section 3.3. Invocation of a Vesta operator primitive within the pseudo-code is denoted by the `operator` syntax. All other operators appearing in the pseudo-code denote the C++ operators.

In these descriptions, the argument types represent the natural domain; the result type is the natural range. In reality, all functions accept arguments of any type, producing *err* for arguments that lie outside the natural domain. For this reason, a function whose specified (natural) result is of type T has an actual result of type `U(T, t_err)`.

Type-checking occurs when primitive functions are called, not before.

3.4.1 Functions on Type `t_bool`

Recall that *true* and *false* are Vesta values, not C++ quantities.

```
t_bool
operator==(t_bool b1, t_bool b2)
```

Returns *true* if *b1* and *b2* are the same, and *false* otherwise.

```
t_bool
```

```
operator!=(t_bool b1, t_bool b2)
  operator!(operator==(b1, b2))

t_bool
operator!(t_bool b) =
{
  int ib = b; // convert to C++ integer
  if (ib) return false; else return true;
}
```

3.4.2 Functions on Type `t_int`

```
t_bool
operator==(t_int i1, t_int i2)
```

Returns *true* if *i1* and *i2* are equal, and *false* otherwise.

```
t_bool
operator!=(t_int i1, t_int i2) =
  operator!(operator==(i1, i2))

t_int
operator+(t_int i1, t_int i2)
```

Returns the integer sum *i1* + *i2* unless it lies outside the implementation-defined range, in which case *err* is returned.

```
t_int
operator-(t_int i1, t_int i2)
```

Returns the integer difference *i1* - *i2* unless it lies outside the implementation-defined range, in which case *err* is returned.

```
t_int
operator-(t_int i) =
  operator-(0, i)

t_int
operator*(t_int i1, t_int i2)
```

Returns the integer product *i1* * *i2* unless it lies outside the implementation-defined range, in which case *err* is returned.

```
t_int
_div(t_int i1, t_int i2)
```

Returns the integer quotient *i1* / *i2* (that is, the floor of the real quotient) unless it lies outside the implementation-defined range, in which case *err* is returned. (*err* is possible only if *i2* is zero or if *i2* is -1 and *i1* is the largest implementation-defined negative number.)

```
t_int
_mod(t_int i1, t_int i2) =
  operator-(i1, operator*(_div(i1,i2), i2))
```

```
t_bool
operator<(t_int i1, t_int i2) =
{
  int ii1 = i1, ii2 = i2; // convert to C++ integers
  if (ii1 < ii2) return true; else return false;
}

t_bool
operator>(t_int i1, t_int i2) =
  operator<(i2, i1)

t_bool
operator<=(t_int i1, t_int i2) =
{
  int ii1 = i1, ii2 = i2; // convert to C++ integers
  if (ii1 <= ii2) return true; else return false;
}

t_bool
operator>=(t_int i1, t_int i2) =
  operator<=(i2, i1)

t_int
_min(t_int i1, t_int i2) =
{ if (operator<(i1, i2)) return i1; else return i2; }

t_int
_max(t_int i1, t_int i2) =
{ if (operator>(i1, i2)) return i1; else return i2; }
```

3.4.3 Functions on Type `t_text`

The first byte of a t_text value has index 0.

```
t_bool
operator==(t_text t1, t_text t2)
```

Returns *true* if *t1* and *t2* are identical byte sequences, and *false* otherwise.

```
t_bool
operator!=(t_text t1, t_text t2) =
  operator!(operator==(t1, t2))

t_text
operator+(t_text t1, t_text t2)
```

Returns the byte sequence formed by appending the byte sequence *t2* to the byte sequence *t1* (concatenation).

```
t_int
_length(t_text t)
```

Returns the number of bytes in the byte sequence *t*.

```
t_text
```

```
_elem(t_text t, t_int i)
```

If `0 <= i < _length(t)`, returns a byte sequence of length 1 consisting of byte *i* of the byte sequence *t*. Otherwise, returns the empty byte sequence.

```
t_text
_sub(t_text t, t_int start = 0, t_int len = _length(t)) =
{
  int w = _length(t);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(t); extract [i..j)
  t_text r = "";
  for (; i < j; i++) r = operator+(r, _elem(t, i));
  return r;
}
```

Extracts from *t* and returns a byte sequence of length *len* beginning at byte *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces *err* only if it is passed arguments of the wrong type.

```
t_int
_find(t_text t, t_text p, t_int start = 0) =
{
  int j = _length(t) - _length(p);
  if (j < 0) return -1;
  int i = _max(start, 0);
  if (i > j) return -1;
  for (; i <= j; i++) {
    int k = 0;
    while (k < _length(p) && _elem(t, i+k) == _elem(p, k)) k++;
    if (k == _length(p)) return i;
  }
  return -1;
}
```

Finds the leftmost occurrence of *p* in *t* that begins at or after position *start*. Returns the index of the first byte of the occurrence, or -1 if none exists.

```
t_int
_findr(t_text t, t_text p, t_int start = 0) =
{
  int j = _length(t) - _length(p);
  if (j < 0) return -1;
  int i = _max(start, 0);
  if (i > j) return -1;
  for (; i <= j; j--) {
    int k = 0;
    while (k < _length(p) && _elem(t, j+k) == _elem(p, k)) k++;
    if (k == _length(p)) return j;
  }
  return -1;
}
```

Finds the rightmost occurrence of *p* in *t* that begins at or after position *start*. Returns the index of

the first byte of the occurrence, or -1 if none exists.

3.4.4 Functions on Type `t_list`

```
t_bool
operator==(t_list l1, t_list l2)
```

Returns *true* if *l1* and *l2* are lists of the same length containing (recursively) equal values, and *false* otherwise.

```
t_bool
operator!=(t_list l1, t_list l2) =
  operator!(operator==(l1, l2))

t_list
_list1(t_value v)
```

Returns a list containing a single element whose value is *v*.

```
t_value
_head(t_list l)
```

Returns the first element of *l*. If *l* is empty, returns *err*.

```
t_list
_tail(t_list l)
```

Returns the list consisting of all elements of *l*, in order, except the first. If *l* is empty, returns *err*.

```
t_int
_length(t_list l)
```

Returns the number of (top-level) values in the list *l*.

```
t_value
_elem(t_list l, t_int i)
```

Returns the *i*-th value in the list *l*, or *err* if no such value exists. The first value of a list has index 0.

```
t_list
operator+(t_list l1, t_list l2)
```

Returns the list formed by appending *l2* to *l1*.

```
t_list
_sub(t_list l, t_int start = 0, t_int len = _length(l))
{
  int w = _length(l);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(l); extract [i..j)
  t_list r = emptylist;
  for (; i < j; i++) r = operator+(r, _elem(l, i));
```

```
  return r;
}
```

Returns the sub-list of *l* of length *len* starting at element *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces *err* only if it is passed arguments of the wrong type.

```
t_list
_map(t_closure f, t_list l) =
{
  t_list res = emptylist;
  for (; !(l == emptylist); l = _tail(l)) {
    t_value v = f(_head(l)); // apply the closure "f"
    if (res == err || v == err) res = err;
    else res = operator+(res, v);
  }
  return res;
}
```

Returns the list that results from applying the closure *f* to each element of the list *l*, and concatenating the results in order. The closure *f* should take one value (of type t_value) as argument and return a value of any type. If *f* has the wrong signature or if any evaluation of *f* returns *err*, then `_map` returns *err*. However, *f* will be applied to every element of the list, even if one of its evaluations produces *err*.

```
t_list
_par_map(t_closure f, t_list l)
```

Formally equivalent to `_map`, but the implementation may perform each application of *f* in a separate parallel thread. External tools invoked by `_run_tool` in different threads may be run simultaneously on different machines.

3.4.5 Functions on type `t_binding`

```
t_bool
operator==(t_binding b1, t_binding b2)
```

Returns *true* if *b1* and *b2* are bindings of the same length containing the same names (in order) bound to (recursively) equal values, and *false* otherwise.

```
t_bool
operator!=(t_binding b1, t_binding b2) =
  operator!(operator==(b1, b2))

t_binding
_bind1(t_text n, t_value v)
```

If *n* is empty, returns *err*. Otherwise, returns a binding with the single <name, value> pair <*n*, *v*>. Note that *v* may be any value, including *err*.

```
t_binding
_head(t_binding b)
```

Returns a binding with one <name, value> pair equal to the first element of *b*. If *b* is empty, returns

err.

```
t_binding
_tail(t_binding b)
```

Returns the binding consisting of all elements of *b*, in order, except the first. If *b* is empty, returns *err*.

```
t_int
_length(t_binding b)
```

Returns the number of <name, value> pairs in *b*.

```
t_binding
_elem(t_binding b, t_int i)
```

Returns a binding consisting solely of the *i*-th <name, value> pair in the binding *b*, or *err* if no such pair exists. The first pair of a binding has index 0.

```
t_text
_n(t_binding b)
```

If `_length(b) = 1`, returns the name part of the <name, value> pair that constitutes *b*. Otherwise, returns *err*.

```
t_value
_v(t_binding b)
```

If `_length(b)` differs from 1, returns *err*. Otherwise, let *v* be the value part of the <name, value> pair that constitutes *b*. This function returns *v*. (Note that a result value of *err* does not imply that `_length(b)` differs from 1, since *v* may be the value *err*.)

```
t_bool
_defined(t_binding b, t_text name)
```

If *name* is empty, returns *err*. Otherwise, returns *true* if the binding *b* contains a pair <n, v> with *n* identical to *name*, and *false* otherwise.

```
t_value
_lookup(t_binding b, t_text name)
```

If *name* is empty, returns *err*. If *name* is defined in *b*, returns the value associated with it; otherwise, returns *err*. Note that the value associated with *name* may be of any type, including t_err, so a result of *err* does not necessarily imply that `_defined(b, name)` is *false*.

```
t_binding
_append(t_binding b1, t_binding b2)
```

Returns a binding formed by appending *b2* to *b1*, but only if all the names in *b1* and *b2* are distinct. Otherwise, returns *err*.

```
t_binding
```

```
operator+(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 == emptybinding); b1 = _tail(b1)) {
    val n = _n(_head(b1));
    val v;
    if (_defined(b2, n) == true)
      v = _lookup(b2, n);
    else v = _v(_head(b1));
    r = _append(r, _bind1(n, v));
  }
  for (; !(b2 == emptybinding); b2 = _tail(b2)) {
    if (_defined(b1, _n(_head(b2)) == false)
      r = _append(r, _head(b2));
  }
  return r;
}
```

Returns a binding formed by appending *b2* to *b1*, giving precedence to *b2* when both *b1* and *b2* contain <name, value> pairs with the same *name*.

```
t_binding
operator++(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 == emptybinding); b1 = _tail(b1)) {
    val n = _n(_head(b1));
    val v;
    if (_defined(b2, n) == true) {
      val v2 = _lookup(b2, n);
      if (_is_binding(v2) == true) {
        v = _v(_head(b1);
        if (_is_binding(v) == true)
          v = operator++(v, v2);
        else v = v2;
      }
      else v = v2;
    }
    else v = _v(_head(b1));
    r = _append(r, _bind1(n, v));
  }
  for (; !(b2 == emptybinding); b2 = _tail(b2)) {
    if (_defined(r, _n(_head(b2)) == false)
      r = _append(r, _head(b2));
  }
  return r;
}
```

Similar to operator+, but performs the operation recursively for each name *n* for which both `_isbinding(_lookup(b1, n))` and `_isbinding(_lookup(b2, n))` are true.

```
t_binding
operator-(t_binding b1, t_binding b2) =
{
  val r = emptybinding;
  for (; !(b1 = emptybinding); b1 = _tail(b1)) {
```

```
    val n = _n(_head(b1));
    if (_defined(b2, n) == false)
      r = _append(r, _head(b1));
  }
  return r;
}
```

Returns a binding formed by removing from *b1* any pair <*n*, *v*> such that `_defined(b2, n)`. The value *v* associated with *n* in *b2* is irrelevant.

```
t_binding
_sub(t_binding b, t_int start = 0, t_int len = _length(b))
{
  int w = _length(b);
  int i = _min(_max(start, 0)), w);
  int j = _min(i + _max(len, 0), w);
  // 0 <= i <= j <= _length(b); extract [i..j)
  t_binding r = emptybinding;
  for (; i < j; i++) r = _append(r, _elem(b, i));
  return r;
}
```

Returns the sub-binding of *b* of length *len* starting at element *start*. Note the boundary cases defined by the pseudo-code; `_sub` produces *err* only if it is passed arguments of the wrong type.

```
t_binding
_map(t_closure f, t_binding b) =
{
  t_binding res = emptybinding;
  for (; !(b == emptybinding); b = _tail(l)) {
    t_binding b1 = f(_n(_head(b)), _v(_head(b))); // apply the closure "f"
    if (res == err || b1 == err) res = err;
    else res = _append(res, b1);
  }
  return res;
}
```

Returns the binding that results from applying the closure *f* to each <*name*, *value*> pair of the binding *b*, and appending the resulting bindings together. The closure *f* should take the *name* (of type t_text) and *value* (of type t_value) as arguments, and return a value of type t_binding. If *f* has the wrong signature or if any evaluation of *f* returns *err*, then `_map` returns *err*. However, *f* will be applied to every pair of the binding, even if one of its evaluations produces *err*.

```
t_binding
_par_map(t_closure f, t_binding b)
```

Formally equivalent to `_map`, but the implementation may perform each application of *f* in a separate parallel thread. External tools invoked by `_run_tool` in different threads may be run simultaneously on different machines.

3.4.6 Type Manipulation Functions

```
t_text
_type_of(t_value v)
```

`_type_of` returns a text value corresponding to the type of the value *v*:

```
  value             text returned by _type_of
  -----             -------------------------
  true, false       "t_bool"
  integer           "t_int"
  byte sequence     "t_text"
  err               "t_err"
  list              "t_list"
  binding           "t_binding"
  closures          "t_closure"
```

```
t_bool
_same_type(t_value v1, t_value v2) =
   operator==(_type_of(v1), _type_of(v2))

t_bool
_is_bool(t_value v)
```

Returns *true* if *v* is of type t_bool; returns *false* otherwise.

```
t_bool
_is_int(t_value v)
```

Returns *true* if *v* is of type t_int; returns *false* otherwise.

```
t_bool
_is_text(t_value v)
```

Returns *true* if *v* is of type t_text; returns *false* otherwise.

```
t_bool
_is_err(t_value v)
```

Returns *true* if *v* is of type t_err; returns *false* otherwise.

```
t_bool
_is_list(t_value v)
```

Returns *true* if *v* is of type t_list; returns *false* otherwise.

```
t_bool
_is_binding(t_value v)
```

Returns *true* if *v* is of type t_binding; returns *false* otherwise.

```
t_bool
_is_closure(t_value v)
```

Returns *true* if *v* is of type t_closure; returns *false* otherwise.

3.4.7 Tool Invocation Function

```
t_binding
_run_tool(
  platform: t_text,
  command:  t_list,
  stdin:    t_text = "",
  stdout_treatment: t_text = "report",
  stderr_treatment: t_text = "report",
  status_treatment: t_text = "report_nocache",
  signal_treatment: t_text = "report_nocache",
  fp_contents: t_int = 0,
  wd: t_text = ".WD",
  existing_writable: t_bool = FALSE)
```

`_run_tool` is the mechanism by which external programs like compilers and linkers are executed from a Vesta SDL program. It provides functionality that is fairly platform-independent. The following description, however, is somewhat Unix-specific (for example, in its description of exit codes and signals).

The *platform* argument specifies the platform on which the tool is to be executed. `_run_tool` selects a specific machine for the given platform. The legal values for *platform* and the mechanism by which a machine of the appropriate platform is chosen are implementation dependent.

The tool to be executed is specified by the *command* argument. This argument is a t_list of t_text values. The first member of the list is the name of the tool (interpretation of the name is discussed below); the remaining members of the list are the arguments passed to the tool as its command line. The tool is executed on the specified *platform* in an environment with the following characteristics:

- The file system is encapsulated so that absolute paths (i.e., those beginning with a Delim) are interpreted relative to `./root`, where `.' is the implicit final parameter to _run_tool. Non-absolute paths are interpreted relative to `./root/$`*`wd`*, where *wd* is a parameter to _run_tool. The interpretation of filenames is discussed in more detail below.

- The environment variables are taken from `./envVars`, where `.' is the implicit final parameter to _run_tool.

- The contents of standard input are the value of the *stdin* parameter to _run_tool.

- Standard output and standard error are treated as specified by the *stdout_treatment* and *stderr_treatment* parameters. Each of these parameters may take on one of the t_text values `"ignore"`, `"report"`, `"report_nocache"`, or `"value"`. If the value is `"ignore"`, any bytes written to the corresponding output stream (stdout or stderr) are discarded. If the value is `"report"`, the corresponding output is made visible to the user. If the value is `"report_nocache"`, the corresponding output is made visible to the user and, if it is not empty, the evaluator does not cache the _run_tool result. If the value is `"value"`, the output stream is converted to a Vesta value of type t_text and returned as part of the _run_tool result, as described below.

- The *status_treatment* and *signal_treatment* arguments may take on the t_text value `"report"` or `"report_nocache"`. Regardless of their values, the `code` and `signal` fields of the result value will be set as described below. If the value of *status_treatment* is

`"report_nocache"`, this run_tool call will not be cached if the result `code` is nonzero; similarly, if *signal_treatment* is `"report_nocache"`, the run_tool call will not be cached if the result `signal` is nonzero.

- The *existing_writable* argument controls whether the tool is permitted to write to files that already exist in its encapsulated file system when it is started. If the argument is TRUE, such files may be opened for writing and written to; if it is FALSE, they may not. For technical reasons in the NFS-based repository implementation, tools will get much better file system performance when *existing_writable* is FALSE. It should be set to TRUE only for tools that require it.

In the absence of errors, _run_tool returns a binding that contains the results of the command execution. This binding has type:

```
type run_tool_result = binding [
  code   : int,
  signal : int,
  stdout_written : bool,
  stderr_written : bool,
  stdout : text,
  stderr : text,
  root   : binding
]
```

If *r* is of type run_tool_result, then:

- *r*`/code` is an integer value that characterizes how the command terminated (i.e., the exit status of the Unix process).
- *r*`/signal` is an integer value identifying the Unix signal that terminated the process, or 0 if the process exited voluntarily.
- *r*`/stdout_written` and *r*`/stderr_written` indicate whether data was written to the stdout and stderr streams, respectively.
- *r*`/stdout` is defined iff the *stdout_treatment* parameter to _run_tool is `"value"`, in which case it contains the bytes written to stdout.
- *r*`/stderr` is defined iff the *stderr_treatment* parameter to _run_tool is `"value"`, in which case it contains the bytes written to stderr.
- *r*`/root` is a binding containing all files created by the command that are extant upon exit. See File System Encapsulation below for more details.

Two fine points relating to the results of _run_tool:

1. If the tool cannot be invoked---for example, because of errors in the parameters to _run_tool---the evaluator prints a suitable diagnostic and the _run_tool call returns *err*. However, errors that result during the execution of the tool are reported in a tool-specific fashion, with the exit status reported in *r*`/code`.

2. Specifying `"report_nocache"` as the treatment for an output stream (stdout or stderr) or the exit status prevents the evaluator from making a cache entry from the call of _run_tool if any output is produced on the corresponding output stream or if the exit status is nonzero, respectively. In addition, none of the ancestor functions of the failing _run_tool call in the call graph are cached either. Since no cache entries are made, a subsequent re-interpretation of the model will produce the same output (on stdout or stderr). This can be useful for reproducing error messages from a compiler or other external tool that are displayed through the Vesta user interface.

By default, arbitrary unique fingerprints are chosen for any derived files created by the tool execution, including derived files created for stdout/stderr when the value of the *stdout_treatment*/*stderr_treatment* parameter is "value". You can instead cause the fingerprints for such files to be computed deterministically from their contents, using the *fp_contents* parameter. If this parameter is a nonnegative value, files less than *fp_contents* bytes long are given content-based fingerprints, while files of *fp_contents* or more bytes are given arbitrary unique fingerprints. If the parameter is set to -1, all files are given content-based fingerprints. The boolean values TRUE and FALSE are accepted as synonyms for -1 and 0 respectively.

The cost of fingerprinting a file's contents is non-trivial, but doing so allows for cache hits in cases where two evaluations depends on an value that is identical but was computed in two different ways.

File System Encapsulation:

- When the command process (or any subprocess it creates) executes a Unix system call that includes a file path as a parameter, the file path is translated into a reference into the `.' binding that is the last parameter to _run_tool.

- The path is interpreted beginning at `./root` if it begins with "/" and at `./root/$wd` otherwise, where `$wd` is the value of the *wd* parameter to _run_tool. Each component of the path---except possibly the final one---must name a Vesta binding. The interpretation of the final component of the path depends on the semantics of the system call. If the system call expects an extant file, the final component must name a Vesta value of type t_text. If the system call expects an extant directory, the Vesta value must be of type t_binding. If the system call expects an unbound name, the name must not be bound by the binding corresponding to the penultimate path component.

- A file created or modified by the command process (or a subprocess) remains visible in the name space throughout the remainder of the process's execution (or until deleted), just as in a regular file system. This is achieved by modeling file creation, modification, and deletion as a suitable overlaying of `./root`. For example, if the process creates ``foo.o" in its working directory, this has the effect of:

```
./root/$wd += [ foo.o = <bytes of file> ];
<subsequent execution of the command process>
```

- File modification is handled in exactly the same way. For example, if the process opens the

existing file ``foo.db" in its working directory and writes to it, this has the effect of:

```
./root/$wd += [ foo.db = <new contents of file> ];
<subsequent execution of the command process>
```

Note that modification of preexisting files is forbidden if the *existing_writable* argument to _run_tool is set to FALSE (its default value).

- File deletions are modeled similarly, but the files are removed from the context using the binding difference (-) operator, instead of added using the binding overlay (+) operator.

- When the command process exits, the accumulated effects of the file creations and deletions it has performed are returned as part of the _run_tool result (in `r/root`). In this binding, the names of files deleted by the tool are bound to *false*. Such names correspond either to files that existed in `./root` before the tool was invoked, or to files created and subsequently deleted by the tool.

  Thus, if `./root` represents the state of the file system visible to the command process at the time it is launched, then the state of the file system when it exits can be described as:

  ```
  ./root ++ r/root
  ```

  So, if the invoker of _run_tool wanted to update `./root` to reflect the changes made by calling _run_tool, the code might look like this:

  ```
  r = _run_tool( <suitable parameters> );
  new_fs = ./root ++ r/root;
  . += [ root = new_fs ];
  ```

  After the last assignment, names in `./root` bound to *false* are files that were deleted by the tool. Here is a recursive function for removing such files:

  ```
  remove_deleted(b: binding): binding
  {
    res: binding = [];
    foreach [ n = v ] in b do
      res += if v = false then [] else
        if _is_binding(v) then [ $n = remove_deleted(v) ]
        else [ $n = v ];
    return res;
  };
  ```

4. Concrete Syntax

4.1 Grammar

Models:

```
Model       ::= Files Imports Block
```

Files clauses:

```
Files       ::= FileClause*
FileClause  ::= files FileItem*;
FileItem    ::= FileSpec | FileBinding
FileSpec    ::= [ Arc = ] DelimPath
FileBinding ::= Arc = `[' FileSpec*, `]'
```

Import clauses:

```
Imports     ::= ImpClause*
ImpClause   ::= ImpIdReq | ImpIdOpt
ImpIdReq    ::= import ImpItemR*;
ImpItemR    ::= ImpSpecR | ImpListR
ImpSpecR    ::= Arc = DelimPath
ImpListR    ::= Arc = `[' ImpSpecR*, `]'
ImpIdOpt    ::= from DelimPath import ImpItemO*;
ImpItemO    ::= ImpSpecO | ImpListO
ImpSpecO    ::= [ Arc = ] Path [ Delim ]
ImpListO    ::= Arc = `[' ImpSpecO*, `]'
```

Paths and Arcs:

```
DelimPath   ::= [ Delim ] Path [ Delim ]
Path        ::= Arc { Delim Arc }*
Arc         ::= Id | Integer | Text
```

Blocks and Statements:

```
Block       ::= `{' Stmt*; Result; `}'
Stmt        ::= Assign | Iterate | FuncDef | TypeDef
Result      ::= { value | return } Expr
```

Assignment statements:

```
Assign      ::= TypedId [ Op ] = Expr
Op          ::= AddOp | MulOp
AddOp       ::= +  |  ++  |  -
MulOp       ::= *
```

Iteration statements:

```
Iterate     ::= foreach Control in Expr do IterBody
Control     ::= TypedId | `[' TypedId = TypedId `]'
IterBody    ::= Stmt | `{' Stmt+; `}'
```

Function definitions:

```
FuncDef     ::= Id Formals+ [ TypeQual ] Block
Formals     ::= ( FormalArgs )
FormalArgs  ::= { TypedId*,                                    // none defaulted
              | { TypedId = Expr }*,                           // all defaulted
              | TypedId { , TypedId }* { , TypedId = Expr }+ } // some defaulted
```

Expressions:

```
Expr        ::= if Expr then Expr else Expr | Expr1
Expr1       ::= Expr2 {  =>  Expr2 }*
Expr2       ::= Expr3 {  ||  Expr3 }*
Expr3       ::= Expr4 {  &&  Expr4 }*
Expr4       ::= Expr5 [ { == | != | < | > | <= | >= } Expr5 ]
Expr5       ::= Expr6 { AddOp Expr6 }*
Expr6       ::= Expr7 { MulOp Expr7 }*
Expr7       ::= [ UnaryOp ] Expr8
UnaryOp     ::= -  |  !
Expr8       ::= Primary [ TypeQual ]
Primary     ::= ( Expr ) | Literal | Id | List
              | Binding | Select | Block | FuncCall
```

Binary operators with equal precedence are left associative.

Literals:

```
Literal     ::= ERR | TRUE | FALSE | Text | Integer
```

Lists:

```
List        ::= < Expr*, >
```

Bindings:

```
Binding     ::= `[' BindElem*, `]'
BindElem    ::= SelfNameB | NameBind
SelfNameB   ::= Id
NameBind    ::= GenPath = Expr
GenPath     ::= GenArc { Delim GenArc }* [ Delim ]
GenArc      ::= Arc | $ Id | $ ( Expr ) | % Expr %
```

Binding selections:

```
Select      ::= Primary Selector GenArc
Selector    ::= Delim | !
```

Function calls:

```
FuncCall    ::= Primary Actuals
Actuals     ::= ( Expr*, )
```

Type definitions:

```
TypeDef     ::= type Id = Type
TypedId     ::= Id [ TypeQual ]
TypeQual    ::= : Type
Type        ::= any | bool | int | text
              | list [ ( Type ) ]
              | binding ( TypeQual )
              | binding [ ( TypedId*, ) ]
              | function { ( TypedForm*, ) }* [ TypeQual ]
              | Id
```

```
TypedForm   ::= [ Id : ] Type
```

4.2 Ambiguity Resolution

The grammar as given above is ambiguous. We resolve the ambiguity as follows.

The Vesta parser accepts a modified grammar in which the > token is replaced by two distinct tokens: GREATER in the production for Expr4 and RANGLE in the production for List. The modified grammar is unambiguous and can easily be parsed by an LL(1) or LALR(1) automaton.

The Vesta tokenizer is responsible for disambiguating between GREATER and RANGLE wherever > appears in the input. It does so by looking ahead to the next token after the >. If the next token is one of

```
    - ! ( ERR TRUE FALSE Text Integer Id < [ {
```

then the > is taken as GREATER; otherwise, it is taken as RANGLE.

Why is this solution reasonable? Inspection of the grammar shows that in a syntactically valid program, the next token after GREATER must be one of those in the above list. The next token after RANGLE must be one of the following:

```
    : * + ++ - == != < GREATER <= >= && || =>
    ; do , ) then else RANGLE ] % / \ ! (
```

These sets overlap in the tokens `-`, `!`, `(`, and `<`. Because we have chosen to resolve these cases as GREATER, it is impossible to write certain syntactically valid programs containing RANGLE. However, any such program can be rewritten by replacing every List nonterminal by `(` List `)`, yielding a semantically equivalent program in which the closing > of the List is correctly resolved as RANGLE. Moreover, we claim (without presenting a proof) that any program in which RANGLE is followed by `-`, `!`, `(`, or `<` must have a runtime type error, due to the paucity of operators defined on the list type, so in practice such programs are never written.

4.3 Tokens

Here is a BNF description of the tokens of the language. The token classes Delim, Integer, Id, and Text, and the individual tokens in the classes Punc, TwoPunc, and Keyword, serve as terminals in the BNF of earlier sections.

```
TokenSeq   ::= Token*
Token      ::= Integer | Id | Text | Punc | TwoPunc | Keyword
             | Whitespace | Comment

Delim      ::= /  |  \

Integer    ::= DecimalNZ Decimal* | 0 Octal* | 0 { x | X } Hex+
Decimal    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
DecimalNZ  ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Octal      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Hex        ::= Decimal | A | B | C | D | E | F | a | b | c | d | e | f

Id         ::= { Letter | Decimal | IdPunc }+
Letter     ::= 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
             | 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
IdPunc     ::= .  |  _

Text       ::= " TextChar* "
TextChar   ::= Decimal | Letter | Punc | Escape
Punc       ::= ~ | ` | ! | @ | # | $ | % | ^ | & | * | ( | )
            | _ | - | + | = | `{' | `[' | `}' | `]' | : | ;
            | `|' | ' | , | < | . | > | ? | / | Space
Escape     ::= \ { n | t | v | b | r | f | a | \  | " | Octals |  Hexes }
Octals     ::= Octal [ Octal [ Octal ] ]
Hexes      ::= { x | X } Hex [ Hex ]

TwoPunc    ::= ++ | == | != | <= | >= | => | || | &&

Keyword    ::= binding | do | else | ERR | FALSE | files | foreach
            | from | function | if | in | import | list | return
            | then | type | TRUE | value

Whitespace ::= ` ' | Tab | Newline

Comment    ::= // NonNewlineChar* Newline
            |  `/*' CommentBody `*/'
```

We define Newline as an ASCII new line sequence, either CR, LF, or CRLF. NonNewlineChar is any ASCII character other than CR and LF. CommentBody is any sequence of ASCII characters that does not contain `*/'. Tab is the ASCII TAB character.

The ambiguities in the token grammar are resolved as follows. The tokenizer interprets the program as a TokenSeq. It scans from left to right, repeatedly matching the longest possible Token beginning with the next unmatched character. Whitespace and Comment tokens are discarded after matching; other tokens are passed on for parsing by the main grammar. When a string of characters matches both Integer and Id, it is tokenized as Integer. When a string matches both Keyword and Id, it is tokenized as Keyword.

4.4 Reserved Identifiers

Here are Vesta-2's reserved identifiers; they should not be redefined:

```
_append _bind1 _defined _div _elem _find _findr
_head _is_binding _is_bool _is_closure _is_err
_is_int _is_list _is_text _length _list1 _lookup
_map _max _min _mod _n _run_tool _same_type _sub
_tail _type_of _v
```

5. Acknowledgments

Bill McKeeman encouraged us to revise the syntax of the language to make it more palatable to C programmers. Mark Lillibridge gave us many useful comments on an earlier draft of the paper.

6. References

[1] Allan Heydon, Roy Levin, Tim Mann, and Yuan Yu. *The Vesta-2 Software Configuration Management System*, Research Report, Digital Systems Research Center. In preparation.

[2] Roy Levin and Paul R. McJones. *The Vesta Approach to Precise Configuration of Large Software Systems,* Research Report 105, Digital Systems Research Center. June 1993. 39 pgs.

[3] Sheng-Yang Chiu and Roy Levin. *The Vesta Repository: A File System Extension for Software Development*, Research Report 106, Digital Systems Research Center. June 1993. 34 pgs.

[4] Christine B. Hanna and Roy Levin. *The Vesta Language for Configuration Management*, Research Report 107, Digital Systems Research Center. June 1993. 62 pgs.

[5] Mark R. Brown and John R. Ellis. *Bridges: Tools to Extend the Vesta Configuration Management System*, Research Report 108, Digital Systems Research Center. June 1993. 43 pgs.