V Syntax and Documentation

Arthur Giesel Vedana

May 11, 2017

Contents

Introduction

The *V* programming language is a functional language with eager left-to-right evaluation. It has a simple I/O system supporting only direct string operations. It is a trait based strongly and statically typed language supporting both explicit and implicit typing.

This document both specifies the *V* language and shows its implementation in F#. It is divided into 2 categories:

1. Abstract Syntax and Semantics

   This defines the abstract syntax and semantics of the language. It only contains the bare minimum for the language to function, without any syntactic sugar.

2. Language Guide

   This is the actual syntax when programming in *V*. This defines all operators, syntactic sugar and other aspects of the language.

3. Standard Library

   Describes all functions provided in the *V* standard library.

4. Changelog

   Lists the changes done to the language in each version.

1 Abstract Syntax and Semantics

1.1 Abstract Syntax

1.1.1 Expressions

Programs in V are expressions. Each expression is a member of the abstract syntax tree defined below. The syntax tree will be constructed in parts, with an explanation of what each expression means and their uses. The full syntax tree can be obtained by simply joining all the separate sections.

Constants and Variables V has support for a few basic constants and variables.

$$
\begin{array}{lcl}
e & ::= & n \\
 & | & b \\
 & | & c \\
 & | & x \\
 & & \\
x & ::= & \{x_0, x_1, \ldots\} \\
 & & \\
b & ::= & true \mid false \\
n & ::= & \mathbb{Z} \\
c & ::= & `char' \\
char & ::= & \texttt{ASCII characters}
\end{array}
$$

The constants available for the language are:

- Booleans
- Integers
- Characters

For variables, one can define any number of identifiers to be used throughout the program. These variables must be associated with other terms in some way (functions, `let` declarations, etc) as they cannot be evaluated on their own.

Tuples Tuples are ordered heterogeneous collections of values with random access. This means that any element of a tuple can be accessed regardless of its position. Unlike lists, it is not possible to add a value to an existing tuple. Tuples of any size greater than or equal to 2 are supported, and they are of fixed-length after construction.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & (e_1, \ \ldots \ e_n) & (n \geq 2) \\
 & | & \#n \ e &
\end{array}
$$

The first expression is the only way to construct a new tuple in V.
The second expression is the projection of a specific field of a tuple.

Records Records are unordered heterogeneous collection of named values with random access. Each value has a label associated to it, and this label is used to access its associated value. Records of any size greater than 0 are accepted, and their fields are fixed after construction.

The set of labels l is totally ordered, with $l_i < l_j \iff i < j$. For any given record $\{l_1 : e_1, \ \dots \ l_n : e_n\}$, it is guaranteed that $l_i < l_j$ for every $i < j$. This is done to ensure consistent label order in records with the same labels, and also gives an intuitive order in which to evaluate the expression in the operational semantics.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
  & | & \{l_1 : e_1, \ \dots \ l_n : e_n\} & (n \geq 1) \\
  & | & \#l \ e & \\
 & & & \\
l & ::= & \{l_1, l_2, ...\} &
\end{array}
$$

The first expression is the only way to construct a new record in V.

The second expression is the projection of a specific field of a record.

Lists The V language has built-in lists. Each list is an ordered homogeneous finite-length collection of values, meaning that a list contains only elements of the same type. There are also basic operations on lists, such as appending, obtaining the first element of a list, etc.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
  & | & nil \\
  & | & e_1 :: e_2
\end{array}
$$

The first expression is the empty list. It is the only zero-length list possible, and all other lists are constructed on top of it.

The second expression is the append operation, adding e_1 to the front of the list e_2.

Conditional Like most functional languages, V provides a conditional expression. This expression, like all others, always returns a value.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
  & | & \texttt{if} \ e_1 \ \texttt{then} \ e_2 \ \texttt{else} \ e_3
\end{array}
$$

Binary Operations V provides a series of binary operations built-in into the language.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
  & | & e_1 \ op \ e_2 \\
 & & \\
op & ::= & opNum \mid opEq \mid opIneq \\
opNum & ::= & + \mid - \mid * \mid \div \\
opEq & ::= & = \mid \neq \\
opIneq & ::= & < \mid \leq \mid > \mid \geq
\end{array}
$$

They are divided into separate categories, each one requiring terms of specific types. The following list describes the requirements and meaning of each category of built-in operators:

- Numerical

  These operators require numbers and also return numbers.

- Equality

  These operators ($=, \neq$) compare two different values for equality, returning a boolean. The values type must conform to the Equatable trait.

- Inequality

  These operators compare two different values for order, returning a boolean. The values type must conform to the Equatable and Orderable trait.

- Boolean Operators

  These operators perform logical operations on boolean values.

Let and Patterns The `let` expression is used to bind values to identifiers for sub-expressions. The identifier can then be used in sub-expressions to increase readability and reduce repetition when writing code.

`let` expressions can also be used to 'unpack' compound values, such as lists, tuples and records. This is done by using patterns, which can be either typed or untyped.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \\
 & & & \\
p & ::= & patt & \\
 & | & patt : T & \\
 & & & \\
patt & ::= & x & \\
 & | & \_ & \\
 & | & n & \\
 & | & b & \\
 & | & c & \\
 & | & nil & \\
 & | & p_1 :: p_2 & \\
 & | & (p_1,\ \ldots\ p_n) & (n \geq 2) \\
 & | & \{l_1 : p_1,\ \ldots\ l_n : p_n\} & (n \geq 1)
\end{array}
$$

This expression works by replacing all occurrences of x by the value of e_1 in e_2 and then evaluating the resulting expression. (In reality, this replacement only occurs as needed to be more efficient (see 1.2), but the result is the same).

The patterns x and _ match any value, with the difference that the second does not bind the value to any identifier. This can be used when a value is not wanted, such as an element of a collection.

The patterns n, b and c match the equivalent constants for numbers, booleans and characters, respectively.

The pattern nil matches only the empty list, while $p_1 :: p_2$ matches any list with at least one element (p_1 matches the head of the list, while p_2 matches the tail).

The pattern $(p_1, \ldots p_n)$ matches a tuple with exactly n elements, and $\{l_1 : p_1, \ldots l_n : p_n\}$ matches a record with fields $l_1, \ldots l_n$.

Match Expression This expression attempts to match a value against multiple patterns, returning the correponding result expression if a match is found.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n & (n \geq 1) \\
 & & & \\
match & ::= & p \rightarrow e & \\
 & | & p\ \texttt{when}\ e_1 \rightarrow e_2 &
\end{array}
$$

A match expression can have any non-zero number of patterns, and they are tested from top to bottom. The first pattern that succesfully matches the input expression interrupts the matching process, and its corresponding result expression is returned.

It is possible to add an extra condition that needs to be satisfied to a pattern by using the `when` keyword, followed by a test expression. This expression has access to any variables declared in its pattern, and is only evaluated if the matching succeeds.

Functions The expressions below all relate to function and function application in the V language.

$$
\begin{array}{lcl}
e & ::= & \cdots \\
 & | & \texttt{fn}\ p \Rightarrow e \\
 & | & \texttt{rec}\ x : T\ \ p \Rightarrow e \\
 & | & \texttt{rec}\ x\ p \Rightarrow e \\
 & | & e_1\ e_2
\end{array}
$$

The first expression defines a simple unnamed function that takes exactly one parameter, p. When a value v is applied to a function, all occurrences of x in e are replaced by v and then the expression is evaluated.

The following two expressions define recursive functions that also take one parameter. In these expressions, x is the name of the function that can be recursively called inside e. p is the single parameter for the function. Recursive functions have two variations: explicitly and implicitly typed. In the explicitly typed variation, the type T is the return type of the function.

In all function expressions, the parameter is declared as a pattern. This means that the function can have multiple identifiers to address the elements of its parameter in the case that it is a collection.

The last expression is the application of e_2 to a function e_1.

Exceptions This expression always evaluates to a runtime error.

$$\begin{array}{lll} e & ::= & \cdots \\ & | & raise \end{array}$$

Runtime errors usually happen when an expression cannot be correctly evaluated, such as division by zero, accessing an empty list, etc.

Sometimes, however, it can be necessary to directly cause an error. The `raise` expression serves this purpose.

1.1.2 Types

Since V is strongly typed, every (valid) expression has exactly one type associated with it. Some expressions require the programmer to explicitly declare types of identifiers, such as `let` declarations and functions. Other expressions, such as $e_1 = e_2$, or even constants, such as 1 or *true*, have types implicitly associated with them. These types are used by the type system (see 1.3) to check whether an expression is valid or not, avoiding run-time errors that can be detected at compile time.

Base Types These are all of the base types available in V. They are considered constant, and cannot be deconstructed or replaced by other types.

$$\begin{array}{lll} T & ::= & \text{Int} \\ & | & \text{Bool} \\ & | & \text{Char} \end{array}$$

Parametric Types These types are composed of 1 or more other types. If the argument types of a parametric type are constant, we can say that the parametric type itself is also constant. This means that 'Int *list*' is a constant type, while '$VarType_1 \rightarrow Int$' is not.

$$\begin{array}{llll} T & ::= & \cdots & \\ & | & T_1 \rightarrow T_2 & \\ & | & T\ list & \\ & | & (T_1,\ \ldots\ T_n) & (n \geq 2) \\ & | & \{l_1 : T_1,\ \ldots\ l_n : T_n\} & (n \geq 1) \end{array}$$

Variable Types These types represent an unknown constant type. Explicitly typed expressions cannot be given variable types, but they are used by the type system for implicitly typed expressions. In the course of the type inference, the type system can replace variable types for their corresponding parametric or constant type.

It is important to realize that variable types already represent a unique type with an unknown identity. This means that a variable type may only be replaced by the specific type which it represents and not any other type. This distinction becomes important when talking about polymorphism, which uses variable types, along with universal quantifiers, to represent a placeholder for any possible type (this is discussed in greater detail in 1.3.1).

$$
\begin{array}{lcl}
T & ::= & \cdots \\
 & | & X^{Traits}
\end{array}
$$

$$
\begin{array}{lcl}
X & ::= & X_1, X_2, ...
\end{array}
$$

1.1.3 Traits

Types can possess traits, which define certain behaviors that are expected of said type. Constant types always have their trait information implicitly defined, since this information is included in the language. Variable types, on the other hand, can explicitly state which traits they possess, restricting the set of possible constant types they can represent.

$$
\begin{array}{lcl}
Traits & ::= & \emptyset \\
 & | & \{Trait\} \cup Traits
\end{array}
$$

$$
\begin{array}{lcll}
Trait & ::= & Equatable & \\
 & | & Orderable & \\
 & | & (n : Type) & \text{(Tuple Position)} \\
 & | & \{l : Type\} & \text{(Record Label)}
\end{array}
$$

Equatable If a type T is *Equatable*, expressions of type T can use the equality operators $(=, \neq)$.

To define the set of types that belong to *Equatable*, the following rules are used:

$$
\begin{array}{l}
\{\text{Int}, \text{Bool}, \text{Char}\} \subset Equatable \\
T \in Equatable \implies T\ list \in Equatable \\
X^{Traits} \in Equatable \implies Equatable \in Traits
\end{array}
$$

Orderable If a type T is *Orderable*, expressions of type T can use the inequality operators $(<, \leq, >, \geq)$. Any type that is *Orderable* is also *Equatable*. In practice, the only difference between *Orderable* and *Equatable* is that the base type Bool is not in *Orderable*.

To define the set of types that belong to *Orderable*, the following rules are used:

$$
\begin{array}{l}
\{\text{Int}, \text{Char}\} \subset Orderable \\
T \in Orderable \implies T\ list \in Orderable \\
X^{Traits} \in Orderable \implies Orderable \in Traits
\end{array}
$$

Tuple Position A tuple position trait specifies a type T that a tuple must have at a certain position n. For this to be valid, the tuple must be at least of size $n+1$, since they are 0-indexed. This trait does not put an upper bound on the size of the tuple.

To define the set of types that belong to a tuple position $(n : T)$, the following rules are used:

$$
\begin{array}{l}
(T_1,\ \ldots\ T_{n+1},\ \ldots\ T_k) \in (n : T) \iff T_{n+1} = T \qquad (k \geq 2,\ 0 \leq n < k) \\
X^{Traits} \in (n : T) \implies (n : T) \in Traits
\end{array}
$$

Record Label A record label trait specifies a type T that a record must have associated to a label l. No bounds are placed on the size of the record, since records are unordered sets of label-type pairs.

To define the set of types that belong to a record label $\{l : T\}$, the following rules are used:

$$\{l_1 : T_1,\ \ldots\ l_n : T_n,\ \ldots\ T_k\} \in \{l : T\} \iff l_n = l \wedge T_n = T \qquad (1 \leq n \leq k)$$
$$X^{Traits} \in \{l : T\} \implies \{l : T\} \in Traits$$

1.2 Operational Semantics

The V language is evaluated using a big-step evaluation with environments. This evaluation reduces an expression into a value directly, not necessarily having a rule of evaluation for every possible expression. To stop programmers from creating programs that cannot be evaluated, a type inference system will be specified later.

Value A value is the result of the evaluation of an expression in big-step. This set of values is different from the set of expressions of V, even though they share many similarities.

Environment An environment is a mapping of identifiers to values that is extended each time a *let* declaration in encountered. Every expression must be evaluated before being stored in the environment, which means that V has eager evaluation.

Below are the definitions of both values and environments:

$$
\begin{array}{lcll}
env & ::= & \{\} \mid \{x \rightarrow v\} \cup env & \\
 & & & \\
v & ::= & n & \\
 & \mid & b & \\
 & \mid & c & \\
 & \mid & nil & \\
 & \mid & v_1 :: v_2 & \\
 & \mid & raise & \\
 & \mid & (v_1, \ \ldots \ v_n) & (n \geq 2) \\
 & \mid & \{l_1 : v_1, \ \ldots \ l_n : v_n\} & (n \geq 1) \\
 & \mid & \langle p, e, env \rangle & \\
 & \mid & \langle x, p, e, env \rangle &
\end{array}
$$

The values $\langle p, e, env \rangle$ and $\langle x, p, e, env \rangle$ are closures and recursive closures, respectively. They represent the result of evaluating functions and recursive functions, both and store the environment at the moment of evaluation. This means that V has static scope, since closures capture the environment at the moment of evaluation and V has eager evaluation.

Closures also store the pattern for the parameter of the respective function (as p), along with the function body (as e). Recursive closures, besides storing the pattern for the parameter (as p) and the function body, also store the name of the function (as x). This allows the function to be called inside its own body, something that the simple closure does not allow.

Pattern Matching For `let` expressions and application, it is necessary to match a pattern p to a value v. This process, if successful, creates a mapping of identifiers to their corresponding elements of v. If the process fails, it means that v does not match the pattern p.

There are two ways in which the matching can fail. In the first way, the structure of the pattern is different from the value, such as matching a tuple pattern with a list value. In this case, the code is invalid and evaluation stops.

In the second way, the structures match, but the actual values do not. This happens in lists, when a *nil* pattern is matched against a non-empty list value (or vice-versa), or with constants (n, b and c).

In this case, the code is valid, but it will evaluate to *raise*.

To aid in this matching, a auxiliary "match" function is defined. The function takes a pattern p and a value v, returning a boolean and a mapping. The boolean indicates whether or not the matching was succesful. The failure that it indicates is the second kind, and the first kind of failure is indicated by a lack of return.

The second value that it returns is the mapping of identifiers to values.

The following are the rules for the match function:

$$match(x, v) = true, \{x \to v\}$$

$$match(\_, v) = true, \{\}$$

$$\frac{||n_1|| = ||n_2||}{match(n_1, n_2) = true, \{\}}$$

$$\frac{||n_1|| \neq ||n_2||}{match(n_1, n_2) = false, \{\}}$$

$$\frac{||b_1|| = ||b_2||}{match(b_1, b_2) = true, \{\}}$$

$$\frac{||b_1|| \neq ||b_2||}{match(b_1, b_2) = false, \{\}}$$

$$\frac{||c_1|| = ||c_2||}{match(c_1, c_2) = true, \{\}}$$

$$\frac{||c_1|| \neq ||c_2||}{match(c_1, c_2) = false, \{\}}$$

$$match(nil, v_1 :: v_2) = false, \{\}$$

$$match(nil, nil) = true, \{\}$$

$$match(p_1 :: p_2, nil) = false, \{\}$$

$$\frac{match(p_1, v_1) = false, env_1}{match(p_1 :: p_2, v_1 :: v_2) = false, \{\}}$$

$$\frac{match(p_1, v_1) = true, env_1 \qquad match(p_2, v_2) = false, env_2}{match(p_1 :: p_2, v_1 :: v_2) = false, \{\}}$$

$$\frac{match(p_1, v_1) = true, env_1 \qquad match(p_2, v_2) = true, env_2}{match(p_1 :: p_2, v_1 :: v_2) = true, env_1 \cup env_2}$$

$$\frac{\exists\ i \in [1, n] \quad match(p_i, v_i) = false, env_i}{match((p_1, ...p_n), (v_1, ...v_n)) = false, \{\}}$$

$$\frac{\forall\ i \in [1, n] \quad match(p_i, v_i) = true, env_i}{match((p_1, ...p_n), (v_1, ...v_n)) = true, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\exists\ i \in [1, n] \quad \exists\ j \in [1, n] \quad l_{1i} = l_{2j} \land match(p_i, v_j) = false, env_i}{match(\{l_{11} : p_1, ...l_{1n} : p_n\}, \{l_{21} : v_1, ...l_{2n} : v_n\}) = false, \{\}}$$

$$\frac{\forall\ i \in [1, n] \quad \exists\ j \in [1, n] \quad l_{1i} = l_{2j} \land match(p_i, v_j) = true, env_i}{match(\{l_{11} : p_1, ...l_{1n} : p_n\}, \{l_{21} : v_1, ...l_{2n} : v_n\}) = true, \bigcup_{i=1}^{n} env_i}$$

1.2.1 Big-Step Rules

$$\text{env} \vdash n \Downarrow n \qquad \text{(BS-Num)}$$

$$\text{env} \vdash b \Downarrow b \qquad \text{(BS-Bool)}$$

$$\text{env} \vdash c \Downarrow c \qquad \text{(BS-Char)}$$

$$\frac{\text{env}(x) = v}{\text{env} \vdash x \Downarrow v} \qquad \text{(BS-Ident)}$$

Tuples A tuple construction expression $(e_1,\ \dots\ e_n)$ evaluates each of its sub-expressions individually, resulting in a tuple value. If any expression evaluates to a *raise*, the whole tuple evaluates to *raise*, propagating the exception.

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash (e_1,\ \dots\ e_n) \Downarrow (v_1,\ \dots\ e_n)} \qquad \text{(BS-Tuple)}$$

For field projection, the tuple is 0-indexed, meaning the first component has index 0. Because of this, projection can only be done on positive numbers. Any attempt to project an inexisting field (i.e $n \geq k$) is invalid code, and does not evaluate to *raise*.

$$\frac{\text{env} \vdash e \Downarrow (v_1,\ \dots\ v_k) \qquad 0 \leq \|n\| < \|k\|}{\text{env} \vdash \#n\ e \Downarrow v_{n+1}} \quad \text{(BS-TupleProjection)}$$

Records A record construction expression $\{l_1 : e_1,\ \dots\ l_n : e_n\}$ evaluates each of its sub-expressions individually, resulting in a record value. If any expression evaluates to a *raise*, the whole record evaluates to *raise*, propagating the exception.

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash \{l_1 : e_1,\ \dots\ l_n : e_n\} \Downarrow \{l_1 : v_1,\ \dots\ l_n : v_n\}} \quad \text{(BS-Record)}$$

Labels are used to project a field of a record, and the fields are unordered. This means that, as long as some field of a record has the requested label, that field can be projected. Any attempt to project an inexisting field (i.e the label does not exist in the record) is invalid code, and does not evaluate to *raise*.

$$\frac{\text{env} \vdash e \Downarrow \{l_1 : e_1,\ \dots\ l_n : e_n\} \qquad l = l_k \qquad 1 \leq \|k\| \leq \|n\|}{\text{env} \vdash \#l\ e \Downarrow v_k} \quad \text{(BS-RecordProjection)}$$

Lists The expression *nil* always evaluates to the value *nil*, which represents an empty list. The append operation (::) accepts any value as its first operand (e_1), but the second operand (e_2) must evaluate to either the empty list (*nil*) or a non-empty list (represented by the value $v_1 :: v_2$).

$$\text{env} \vdash nil \Downarrow nil \quad \text{(BS-Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 :: e_2 \Downarrow v :: nil} \quad \text{(BS-List)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 :: e_2 \Downarrow v :: (v_1 :: v_2)} \quad \text{(BS-List2)}$$

Numerical Operations The *V* language only supports integers, so all operations are done on integer numbers. This means that the division always results in a whole number, truncated towards zero.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| + \|n_2\|}{\text{env} \vdash e_1 + e_2 \Downarrow n} \quad \text{(BS-+)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| - \|n_2\|}{\text{env} \vdash e_1 - e_2 \Downarrow n} \quad \text{(BS–)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| * \|n_2\|}{\text{env} \vdash e_1 * e_2 \Downarrow n} \quad (\text{BS-}*)$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow 0}{\text{env} \vdash e_1 \div e_2 \Downarrow raise} \quad (\text{BS-}\div\textsc{Zero})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \\ \|n_2\| \neq 0 \qquad \|n\| = \|n_1\| \div \|n_2\|\end{array}}{\text{env} \vdash e_1 \div e_2 \Downarrow n} \quad (\text{BS-}\div)$$

Equality Operations The equality operators (= and ≠) allow comparison of certain expressions with other expressions of the same kind. In this way, it is a polymorphic operator, being usable in different contexts. Even so, it is important to realize that it only compares values of the same kind (numbers with numbers, characters with characters, etc).

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\| = \|n_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{NumTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\| \neq \|n_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{NumFalse})$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| = \|c_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{CharTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\| \neq \|c_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{CharFalse})$$

$$\frac{\text{env} \vdash e_1 \Downarrow b_1 \qquad \text{env} \vdash e_2 \Downarrow b_2 \qquad \|b_1\| = \|b_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{BoolTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow b_1 \qquad \text{env} \vdash e_2 \Downarrow b_2 \qquad \|b_1\| \neq \|b_2\|}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{BoolFalse})$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{NilTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{NilFalse}1)$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{NilFalse}2)$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow false\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{ListFalse})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow true \qquad \text{env} \vdash v_2 = v_4 \Downarrow b\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow b} \quad (\text{BS-=}\textsc{ListTrue})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow (v_{11}, \ \dots \ v_{1n}) \qquad \text{env} \vdash e_2 \Downarrow (v_{21}, \ \dots \ v_{2k}) \\ \exists\ k \in [1, n] \quad \text{env} \vdash v_{1k} = v_{2k} \Downarrow false \\ \forall\ j \in [1, k) \quad \text{env} \vdash v_{1j} = v_{2j} \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{TupleFalse})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow (v_{11}, \ \dots \ v_{1n}) \qquad \text{env} \vdash e_2 \Downarrow (v_{21}, \ \dots \ v_{2n}) \\ \forall\ k \in [1, n] \quad \text{env} \vdash v_{1k} = v_{2k} \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{TupleTrue})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \{l_1^1 : v_1^1, \ \dots \ l_n^1 : v_n^1\} \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2, \ \dots \ l_n^2 : v_n^2\} \\ \exists\ k \in [1, n] \quad l_k^1 = l_k^2 \wedge \text{env} \vdash v_k^1 = v_k^2 \Downarrow false \\ \forall\ j \in [1, k) \quad \text{env} \vdash v_j^1 = v_j^2 \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow false} \quad (\text{BS-=}\textsc{RecordFalse})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \{l_1^1 : v_1^1, \ \dots \ l_n^1 : v_n^1\} \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2, \ \dots \ l_n^2 : v_n^2\} \\ \forall\ k \in [1, n] \quad l_k^1 = l_k^2 \wedge \text{env} \vdash v_k^1 = v_k^2 \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow true} \quad (\text{BS-=}\textsc{RecordTrue})$$

$$\frac{\text{env} \vdash e_1 = e_2 \Downarrow false}{\text{env} \vdash e_1 \neq e_2 \Downarrow true} \quad (\text{BS-}\neq\textsc{True})$$

$$\frac{\text{env} \vdash e_1 = e_2 \Downarrow true}{\text{env} \vdash e_1 \neq e_2 \Downarrow false} \quad (\text{BS-}\neq\textsc{False})$$

Inequality Operations The inequality operators function much in the same way as the equality operators. The only difference is that they do not allow comparison of certain kinds of expressions (such as booleans) when such expressions do not have a clear ordering to them.

To reduce the number of rules, some rules are condensed for all inequality operators ($<, \leq, >, \geq$). The comparison done on numbers is the ordinary numerical comparison. For characters, the ASCII values are compared numerically.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\|\ opIneq\ \|n_2\|}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow true} \ (\text{BS-IneqNumTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \neg\, \|n_1\|\ opIneq\ \|n_2\|}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow true} \ (\text{BS-IneqNumFalse})$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\|\ opIneq\ \|c_2\|}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow true} \ (\text{BS-IneqCharTrue})$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \neg\, \|c_1\|\ opIneq\ \|c_2\|}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow true} \ (\text{BS-IneqCharFalse})$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \ (\text{BS-<Nil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \ (\text{BS-}\leq\text{Nil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow false} \ (\text{BS->Nil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \ (\text{BS-}\geq\text{Nil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \ (\text{BS-<ListNil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow false} \ (\text{BS-}\leq\text{ListNil})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow \mathit{nil}}{\text{env} \vdash e_1 > e_2 \Downarrow \mathit{true}} \quad \text{(BS->ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow \mathit{nil}}{\text{env} \vdash e_1 \geq e_2 \Downarrow \mathit{true}} \quad \text{(BS-}\geq\text{ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{nil} \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 < e_2 \Downarrow \mathit{true}} \quad \text{(BS-<NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{nil} \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \leq e_2 \Downarrow \mathit{true}} \quad \text{(BS-}\leq\text{NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{nil} \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 > e_2 \Downarrow \mathit{false}} \quad \text{(BS->NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{nil} \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \geq e_2 \Downarrow \mathit{false}} \quad \text{(BS-}\geq\text{NilList)}$$

$$\frac{\begin{array}{cc} \text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow \mathit{false} & \text{env} \vdash v_1 \ \mathit{opIneq} \ v_3 \Downarrow b \end{array}}{\text{env} \vdash e_1 \ \mathit{opIneq} \ e_2 \Downarrow b} \quad \text{(BS-IneqListHead)}$$

$$\frac{\begin{array}{cc} \text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow \mathit{true} & \text{env} \vdash v_2 \ \mathit{opIneq} \ v_4 \Downarrow b \end{array}}{\text{env} \vdash e_1 \ \mathit{opIneq} \ e_2 \Downarrow b} \quad \text{(BS-IneqListTail)}$$

Conditional Expression *V* supports conditional expressions, which always return a value, but not conditional statements. Because of this, all conditional expressions must have both a *then* and an *else* branch. Evaluation is done only on the condition and the proper branch, avoiding the evaluation of the unused branch.

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{true} \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \quad \text{(BS-IfTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{false} \qquad \text{env} \vdash e_3 \Downarrow v}{\text{env} \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \quad \text{(BS-IfFalse)}$$

Function Expressions There are two types of function expressions, one for simple functions and one for recursive functions.

The first is a simple unnamed function that takes exactly one parameter. This parameter can occur anywhere inside the function body (e) and will be replaced by the argument when the function is called.

$$\text{env} \vdash \texttt{fn}\ p \Rightarrow e \Downarrow \langle p, e, env \rangle \tag{BS-Fn}$$

The second type of function is a recursive function that also takes exactly one parameter (p). Unlike the unnamed function, a recursive function also specifies its own name (x), such that it can be called within the function body.

The typed variant must specify the type of the output (T).

$$\text{env} \vdash \texttt{rec}\ x_1 : T \quad p \Rightarrow e \Downarrow \langle x, p, e, env \rangle \tag{BS-Rec}$$

$$\text{env} \vdash \texttt{rec}\ x\ p \Rightarrow e \Downarrow \langle x, p, e, env \rangle \tag{BS-Rec2}$$

Application An application expression requires either a closure or a recursive closure for its left-hand operand. The right-hand operand (argument) is always evaluated using the current environment, resulting in a value v_2.

In the case of a simple closure, the body of the function (e) is evaluated using the stored closure, matching the parameter pattern (p) with the argument (v_2).

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = true, env_1 \\ env_1 \cup \text{env} \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \tag{BS-AppFn}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = false, env_1\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \tag{BS-AppFn2}$$

In the case of a recursive closure, there are two new associations added to the stored closure. The first is, as with a simple closure, the parameter pattern (p) and the argument (v_2). The second is the function identifier (x) and the closure itself. This ensures that the function body can call the recursive function again since its closure is included in the environment.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle x, p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = true, env_1 \\ \{x \to \langle x, p, e, env \rangle\} \cup env_1 \cup \text{env} \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \tag{BS-AppRec}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle x, p, e, env \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ match(p, v_2) = false, env_1\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \tag{BS-AppRec2}$$

If the function does not require the value of its parameter (for example, always returning a constant value), it will evaluate correctly even if its argument is *raise*. This means that V has elements of a non-strict language.

Note, however, that the argument is fully evaluated before being replaced in the function. This means that V also has elements of a strict (or, more accurately, eager) language.

The result of these behaviours is that V is neither fully strict nor fully non-strict. If an unused argument raises an exception, the function behaves as if it is non-strict. If the same function receives an argument whose evaluation does not terminate, it behaves as if it is strict (i.e it also does not terminate).

Let Expressions These expressions are used to associate an identifier with a specific value, allowing the value to be reused throughout the program. Since V is a functional language, these are not variables, and the values assigned to an identifier will be constant (unless the same identifier is used in a new *let* expression).

After evaluating the expression that is to be associated to the identifier (that is, e_1), resulting in v, the *let* expression evaluates e_2. For this evaluation, the association of p to v is added to the environment. The result of this evaluation (that is, v_2) is the final result of the evaluation of the entire *let* expression.

$$\frac{\begin{array}{c}\mathrm{env} \vdash e_1 \Downarrow v \qquad \mathit{match}(p, v) = \mathit{true}, \mathit{env}_1 \\ \mathit{env}_1 \cup \mathrm{env} \vdash e_2 \Downarrow v_2\end{array}}{\mathrm{env} \vdash \mathtt{let}\ p = e_1\ \mathtt{in}\ e_2 \Downarrow v_2} \tag{BS-Let}$$

$$\frac{\mathrm{env} \vdash e_1 \Downarrow v \qquad \mathit{match}(p, v) = \mathit{false}, \mathit{env}_1}{\mathrm{env} \vdash \mathtt{let}\ p = e_1\ \mathtt{in}\ e_2 \Downarrow \mathit{raise}} \tag{BS-Let2}$$

Match Expression The match expression receives a input value and a list of *match*, attempting to pattern match against each one. The first *match* which returns a positive result is considered valid, and its corresponding expression is evaluted as the result of the whole expression.

If no *match* returns a positive result, the whole expression evaluates to *raise*.

$$\frac{\begin{array}{c}\mathrm{env} \vdash e \Downarrow v \\ \exists j \in [1..n]\ \mathit{multiMatch}(v, \mathrm{env}, \mathit{match}_j) = \mathit{true}, v_j \\ \forall k \in [1..j)\ \mathit{multiMatch}(v, \mathrm{env}, \mathit{match}_k) = \mathit{false}, v_k\end{array}}{\mathrm{env} \vdash \mathtt{match}\ e\ \mathtt{with}\ \mathit{match}_1, \ldots\ \mathit{match}_n \Downarrow v_j} \tag{BS-Match}$$

$$\frac{\begin{array}{c}\mathrm{env} \vdash e \Downarrow v \\ \forall j \in [1..n]\ \mathit{multiMatch}(v, \mathrm{env}, \mathit{match}_j) = \mathit{false}, v_j\end{array}}{\mathrm{env} \vdash \mathtt{match}\ e\ \mathtt{with}\ \mathit{match}_1, \ldots\ \mathit{match}_n \Downarrow \mathit{raise}} \tag{BS-Match2}$$

In order to properly evaluate a match expression, it is necessary to define an auxiliary function, here called *multiMatch*. This function receives an input value, an environment and a *match*.

If the *match* has a conditional expression, it must evaluate to *true* for the match to be considered valid.

$$\frac{match(p, v) = false, env_1}{multiMatch(v, \text{env}, p \to e) = false, v}$$

$$\frac{match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e \Downarrow v_2}{multiMatch(v, \text{env}, p \to e) = true, v_2}$$

$$\frac{match(p, v) = false, env_1}{multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2) = false, v}$$

$$\frac{match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e_1 \Downarrow false}{multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2) = false, v}$$

$$\frac{match(p, v) = true, env_1 \qquad \text{env} \cup env_1 \vdash e_1 \Downarrow true \qquad \text{env} \cup env_1 \vdash e_2 \Downarrow v_2}{multiMatch(v, \text{env}, p \text{ when } e_1 \to e_2) = true, v_2}$$

Exceptions Some programs can be syntactically correct but still violate the semantics of the V language, such as a dividing by zero or trying to access the head of an empty list. In these scenarios, the expression is evaluated as the *raise* value.

Besides violation of semantic rules, the only other expression that evaluates to the *raise* value is the *raise* expression, using the following rule:

$$\text{env} \vdash raise \Downarrow raise \qquad \text{(BS-RAISE)}$$

This value is usually ignored, but it propagates upwards if a "regular" value is expected. This means that expressions that need well-defined sub-expressions, such as numerical and equality operations, evaluate to *raise* if any of these sub-expressions evaluate to *raise*.

The rules for *raise* propagation are given below.

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1 e_2 \Downarrow raise} \qquad \text{(BS-APPRAISE)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1 :: e_2 \Downarrow raise} \qquad \text{(BS-LISTRAISE)}$$

$$\frac{\text{env} \vdash e \Downarrow raise}{\text{env} \vdash \#n\ e \Downarrow raise} \qquad \text{(BS-TUPLEPROJECTIONRAISE)}$$

$$\frac{\text{env} \vdash e \Downarrow raise}{\text{env} \vdash \#l\ e \Downarrow raise} \quad (\text{BS-RecordProjectionRaise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=Raise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=Raise2})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow raise\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=ListRaise})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow true \qquad \text{env} \vdash v_2 = v_4 \Downarrow raise\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=ListRaise2})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow (v_{11},\ \dots\ v_{1n}) \qquad \text{env} \vdash e_2 \Downarrow (v_{21},\ \dots\ v_{2n}) \\ \exists\ k \in [1,n] \quad \text{env} \vdash v_{1k} = v_{2k} \Downarrow raise \\ \forall\ j \in [1,k) \quad \text{env} \vdash v_{1j} = v_{2j} \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=TupleRaise})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \{l_1^1 : v_1^1,\ \dots\ l_n^1 : v_n^1\} \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2,\ \dots\ l_n^2 : v_n^2\} \\ \exists\ k \in [1,n] \quad l_k^1 = l_k^2 \wedge \text{env} \vdash v_k^1 = v_k^2 \Downarrow raise \\ \forall\ j \in [1,k) \quad \text{env} \vdash v_k^1 = v_k^2 \Downarrow true\end{array}}{\text{env} \vdash e_1 = e_2 \Downarrow raise} \quad (\text{BS-=RecordRaise})$$

$$\frac{\text{env} \vdash e_1 = e_2 \Downarrow raise}{\text{env} \vdash e_1 \neq e_2 \Downarrow raise} \quad (\text{BS-}\neq\text{Raise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow raise} \quad (\text{BS-IneqRaise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow raise} \quad (\text{BS-IneqRaise2})$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow raise\end{array}}{\text{env} \vdash e_1\ opIneq\ e_2 \Downarrow raise} \quad (\text{BS-IneqListRaise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1 opNum e_2 \Downarrow raise} \quad (\text{BS-NumRaise})$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1 opNum e_2 \Downarrow raise} \qquad \text{(BS-NUMRAISE2)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow raise} \qquad \text{(BS-IFRAISE)}$$

1.3 Type System

V has a strong and static type inference system that checks a program to decide whether or not it is "'well-typed"'. If a program is considered to be well-typed, the type system guarantees that the program will be able to be properly evaluated according to the operational semantics of *V*. As a side-effect of checking the validity of a program, the type system can also provide the actual type of any implicitly typed expression down to its basic types, be those concrete types or variable types.

1.3.1 Polymorphism

V has support for parametric Damas-Milner polymorphism. This means that functions can have their types be defined with universal quantifiers, allowing their use with any type.

For instance, take the function *count*, which counts the number of elements in a list. This function can be defined as follows:

```
let count = rec count x => if isempty x then 0 else 1 + count (tl x) in
count (3::4::nil)
```

In this situation, *count* can be used with a list of any type, not only Int. To allow this, its identifier (*count*) must have a universal association in the environment, defined as so:

$\forall x.\ x\ list \rightarrow \text{Int}$

The universal quantifier $\forall x$ allows the type variable x to be substituted for any concrete type when the function is called. When creating a polymorphic type, the type system must identify which type variables are free in the function type and which are bound in the environment. This process guarantees that a polymorphic type only universally quantifies those type variables that are not already bound, while still allowing all free variables to be instantiated when the function is called.

1.3.2 Traits

Traits are characteristics that a type can have, defining behaviors expected of that type. Some expressions are polymorphic in a sense that they accept certain types for their operators, but not any type.

1.3.3 Type Inference System

The type inference system is composed of two basic parts:

- Constraint Collection

- Constraint Unification

Constraints are equations between type expressions, which can have both constant types and variable types. To infer the type of a program, the type system recursively collects a set of constraints for every subexpression in that program. This is done in a static way across the expression tree from the nodes to the root, without having to evaluate any of the expressions. To create a valid set of constraints, the system must contain an environment, built from the root to the nodes, to ensure identifiers are properly typed.

Environment Just like the operational semantics, the type system also uses an environment to store information about identifiers. In this case, the environment maps identifiers to type associations. These can be either simple associations or universal associations, which are used for polymorphic functions.

Simple Associations These associate an identifier with a unique type, which can be either constant or a variable type. When the association is called, the type is returned as-is, even if it is a variable type.

Universal Associations This association, also called a type scheme, stores a type which contains at least one variable type bound by a "for all" quantifier (∀). When called, this association creates a new variable type for each bound variable and returns a new instance of the type scheme. Universal associations are used exclusively for polymorphic functions.

To create this type of association, the type system must generate a list of "free variables" present in the type that is to be universalized. These are the variable types that are not present in the environment when the identifier is declared. When these free variables are found, they are universally bound. This ensures that only those variable types that are unbound in the environment become universally bound in the resulting association.

Constraint Unification After collecting every type constraint for the program, the type inference system attempts to unify these constraints and find a solution for them. This solution comes in the form of type substitutions, which associate variable types to other types, and type traits, which associate variable types to sets of traits.

If the constraints cannot be unified - that is, if a conflict is found -, the program is deemed not well-typed. Because of how the collection and unification process works, little information is given about where the problem ocurred.

Unification Application After obtaining a valid solution to the set of constraints, the type inference system applies the substitution to the type of the program. This is done recursively until no more substitutions are found, resulting in what is called the principal type. If there are any variable types in the principal type, the traits are applied to them, restricting the set of types that the variable types can represent.

Pattern Matching When a pattern is encountered (such as a `let` expression or function declaration), it is necessary to match the type of the pattern with the value.

To do this, a "match" function is defined. It takes a pattern p and a type T, returning a list of constraints and a mapping of identifiers to associations.

The following are the rules for the "match" function:

$$match(x, T) = \{\}, \{x \to T\}$$

$$match(x : T_1, T_2) = \{T_1 = T_2\}, \{x \to T_1\}$$

$$match(n, T) = \{T = Int\}, \{\}$$

$$match(n : T_1, T_2) = \{T_1 = Int, T_2 = Int\}, \{\}$$

$$match(b, T) = \{T = Bool\}, \{\}$$

$$match(b : T_1, T_2) = \{T_1 = Bool, T_2 = Bool\}, \{\}$$

$$match(c, T) = \{T = Char\}, \{\}$$

$$match(c : T_1, T_2) = \{T_1 = Char, T_2 = Char\}, \{\}$$

$$match(\_, T) = \{\}, \{\}$$

$$match(\_ : T_1, T_2) = \{T_1 = T_2\}, \{\}$$

$$\frac{X_1 \ is \ new}{match(nil, T) = \{X_1 list = T\}, \{\}}$$

$$\frac{X_1 \ is \ new}{match(nil : T_1, T_2) = \{X_1 list = T_1, T_1 = T_2\}, \{\}}$$

$$\frac{X_1 \ is \ new \qquad match(p_1, X_1) = c_1, env_1 \qquad match(p_2, X_1 list) = c_2, env_2}{match(p_1 :: p_2, T) = \{X_1 list = T\} \cup c_1 \cup c_2, env_1 \cup env_2}$$

$$\frac{X_1 \ is \ new \qquad match(p_1, X_1) = c_1, env_1 \qquad match(p_2, X_1 list) = c_2, env_2}{match(p_1 :: p_2 : T_1, T_2) = \{X_1 list = T_1, T_1 = T_2\} \cup c_1 \cup c_2, env_1 \cup env_2}$$

$$\frac{\forall i \in [1, n] \quad X_i \ is \ new \wedge match(p_i, X_i) = c_i, env_i}{match((p_1, ...p_n), T) = \{(X_i, ...X_n) = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge match(p_i, X_i) = c_i, env_i}{match((p_1, ...p_n) : T_1, T_2) = \{(X_i, ...X_n) = T_1, T_1 = T_2\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge match(p_i, X_i) = c_i, env_i}{match(\{l_1 : p_1, ...l_n : p_n\}, T) = \{\{l_1 : X_1, ...l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge match(p_i, X_i) = c_i, env_i}{match(\{l_1 : p_1, ...l_n : p_n\} : T_1, T_2) = \{\{l_1 : X_1, ...l_n : X_n\} = T_1, T_1 = T_2\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} env_i}$$

Constraint Collection Rules Every expression in V has a rule for constraint collection, similar to how every expression has a rule for its semantic evaluation.

$$\Gamma \vdash n : \text{Int} \mid \{\} \tag{T-Num}$$

$$\Gamma \vdash b : \text{Bool} \mid \{\} \tag{T-Bool}$$

$$\Gamma \vdash c : \text{Char} \mid \{\} \tag{T-Char}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T \mid \{\}} \tag{T-Ident}$$

$$\frac{\forall\ k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash (e_1,\ \dots\ e_n) : (T_1,\ \dots\ T_n) \mid C_1 \cup \cdots C_n} \tag{T-Tuple}$$

$$\frac{\Gamma \vdash e : T_1 \mid C_1 \qquad X_1 \textit{ is new} \qquad X_2^{\{(n:X_1)\}} \textit{ is new}}{\Gamma \vdash \#n\ e : X_1 \mid C_1 \cup \{T_1 = X_2\}} \tag{T-TupleProjection}$$

$$\frac{\forall\ k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash \{l_1 : e_1,\ \dots\ l_n : e_n\} : \{l_1 : T_1,\ \dots\ l_n : T_n\} \mid C_1 \cup \cdots C_n} \tag{T-Record}$$

$$\frac{\Gamma \vdash e : T_1 \mid C_1 \qquad X_1 \textit{ is new} \qquad X_2^{\{\{l:X_1\}\}} \textit{ is new}}{\Gamma \vdash \#l\ e : X_1 \mid C_1 \cup \{T_1 = X_2\}} \tag{T-RecordProjection}$$

$$\frac{X_1 \textit{ is new}}{\Gamma \vdash \textit{nil} : X_1 \textit{ list} \mid \{\}} \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 :: e_2 : T_1 \textit{ list} \mid C_1 \cup C_2 \cup \{T_1 \textit{ list} = T_2\}} \quad \text{(T-List)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-+)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 - e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T–)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 * e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-}*\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \div e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-}\div\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Equatable}\}} \textit{ is new}}{\Gamma \vdash e_1 = e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Equatable}\}} = T_2\}} \quad \text{(T-=)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Equatable}\}} \textit{ is new}}{\Gamma \vdash e_1 \neq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Equatable}\}} = T_2\}} \quad \text{(T-}\neq\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Orderable}\}} \textit{ is new}}{\Gamma \vdash e_1 < e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Orderable}\}} = T_2\}} \quad \text{(T-<)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Orderable}\}} \textit{ is new}}{\Gamma \vdash e_1 \leq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Orderable}\}} = T_2\}} \quad \text{(T-}\leq\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Orderable}\}} \textit{ is new}}{\Gamma \vdash e_1 > e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Orderable}\}} = T_2\}} \quad \text{(T->)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\textit{Orderable}\}} \textit{ is new}}{\Gamma \vdash e_1 \geq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{\textit{Orderable}\}} = T_2\}} \quad \text{(T-}\geq\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad \Gamma \vdash e_3 : T_3 \mid C_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \mathrm{Bool} \mid C_1 \cup C_2 \cup C_3 \cup \{T1 = \mathrm{Bool}; T_2 = T_3\}} \text{ (T-If)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1\ \textit{is new}}{\Gamma \vdash e_1\ e_2 : X \mid C_1 \cup C_2 \cup \{T1 = T_2 \rightarrow X_1} \qquad \text{(T-App)}$$

$$\frac{X\ \textit{is new} \qquad \textit{match}(p, X) = C, \textit{env} \qquad \textit{env} \cup \Gamma \vdash e : T_1 \mid C_1}{\Gamma \vdash \texttt{fn}\ p \Rightarrow e : X \rightarrow T_1 \mid C \cup C_1} \qquad \text{(T-Fn)}$$

$$\frac{\begin{array}{c} X\ \textit{is new} \qquad \textit{match}(p, X) = C, \textit{env} \\ \{x \rightarrow (X \rightarrow T)\} \cup \textit{env} \cup \Gamma \vdash e : T_1 \mid C_1 \end{array}}{\Gamma \vdash \texttt{rec}\ x : T \quad p \Rightarrow e : X \rightarrow T_1 \mid C \cup C_1 \cup \{T_1 = T\}} \qquad \text{(T-Rec)}$$

$$\frac{\begin{array}{c} X_1\ \textit{is new} \qquad X_2\ \textit{is new} \qquad \textit{match}(p, X_1) = C, \textit{env} \\ \{x \rightarrow X_2\} \cup \textit{env} \cup \Gamma \vdash e : T_1 | C_1 \end{array}}{\Gamma \vdash \texttt{rec}\ x\ p \Rightarrow e : X_1 \rightarrow T_1 \mid C \cup C_1 \cup \{X_2 = X_1 \rightarrow T_1\}} \qquad \text{(T-Rec2)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T_1 \mid C_1 \qquad \textit{match}(p, T_1) = C, \textit{env} \\ \textit{env} \cup \Gamma \vdash e_2 : T_2 \mid C_2 \end{array}}{\Gamma \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 : T_2 \mid C \cup C_1 \cup C_2} \qquad \text{(T-Let)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T \mid C \qquad X_1\ \textit{is new} \\ \forall j \in [1..n]\ \textit{multiMatch}(T, X_1, \Gamma, \textit{match}_j) = C_j \end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \textit{match}_1, \ldots\ \textit{match}_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i} \qquad \text{(T-Match)}$$

$$\frac{\textit{match}(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e : T_3 \mid C_3}{\textit{multiMatch}(T_1, T_2, \Gamma, p \rightarrow e) = C \cup C_3 \cup \{T_3 = T_2\}}$$

$$\frac{\textit{match}(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e_1 : T_3 \mid C_3 \qquad \Gamma_1 \cup \Gamma \vdash e_2 : T_4 \mid C_4}{\textit{multiMatch}(T_1, T_2, \Gamma, p\ \text{when}\ e_1 \rightarrow e_2) = C \cup C_3 \cup C_4 \cup \{T_3 = \textit{Bool}, T_4 = T_2}$$

$$\frac{X_1\ \textit{is new}}{\Gamma \vdash \textit{raise} : X_1 \mid \{\}} \qquad \text{(T-Raise)}$$

2 Language Guide

2.1 Basic Values

There are 4 types of basic values available in the *V* language:

1. Integers
2. Booleans
3. Character
4. Strings

Integers Only positive integers (plus zero) are recognized. They are always specified in decimal format, using only the digits from 0 to 9.

Booleans Two values are available: `true` for true, and `false` for false.

Characters and Strings A character literal is a single Unicode character surround by single quotes ('). A string literal is a sequence of zero or more Unicode characters surrounded by double quotes ("). Technically, strings are not basic values, since they are just syntactic sugar for a list of characters.
$''abc'' \rightarrow' a' ::' b' ::' c' :: nil$

Some characters must be escaped in order to insert them into either character or string literals. "Escaping" a character means preceding it by the backslash character (\). For character literals, the single quote must be escaped (\'), while string literals require the escaping of the double quote (\").

There is also support for ASCII escape codes to insert special characters in literals. These are the allowed escape codes and their resulting characters:

Escape code	**Character**
\b	backspace
\n	newline (line feed
\r	carriage return
\t	horizontal tab
\\	backslash
\'	single quote
\"	double quote

Any escape code can be used in either character or string literals. Furthermore, the special characters can be inserted directly into the literal. This means that multi line strings are supported by *V*. This also means that a single quote followed by a new line and a single quote is interpreted as a valid character literal (i.e. '\n').

2.2 Compound Values

2.2.1 Lists

Lists are ordered collections of values of the same type. There are no limits on the size of a list, even accepting lists with 0 values (the empty list).

Creating Lists An empty list can be created using either the `nil` keyword or the empty list literal, which is written as `[]` (empy square brackets).

To create a list with values, simply enclose the sequence of values, each separated by a comma, between square brackets.

```
[] // Empty list
[1, 2, 3] // List containing 3 values
```

Expanding Lists It is possible to add a value to the start of a list by using the list construction operator (`::`). The `append` function allowing the addition of a value to the end of a list. It is also possible to create a new list by using the concatenation operator (`@`), which adds two lists together.

```
let x = 0 :: [1, 2, 3];
// x is equal to [0, 1, 2, 3]

let y = append 4 [1, 2, 3];
// y is equal to [1, 2, 3, 4]

let z = [1, 2] @ [3, 4];
// z is equal to [1, 2, 3, 4]
```

Accessing Lists Any element of a list can be accessed by using the index (`!!`) operator. Lists are 0-indexed, which means the first value of a list is at index 0.

```
["a", "b", "c"] !! 0 // Returns "a"
```

An attempt to access an index outside the range of a list (that is, indexes equal to or greater than the size of the list) will result in a runtime error.

```
["a", "b", "c"] !! 5 // Runtime error
```

There are many other operations available for accessing elements of a list, including `head` (returns the first value of a list), `last`, `filter`, `maximum`, etc.

Complex Operations Although the *V* language does not directly support complex operations on lists, the standard library (see 3) provides a number of functions to manipulate lists. Among these are functions like `map`, `filter`, `sort`, `fold`, `sublist`, which provide basic funcionality for performing computations with lists.

Ranges Ranges allow the easy creation of lists of integers in a arithmetic progression.

There are two variants of ranges, one for simple integer counting and one for more complex progressions.

The first variant specifies the first and last value for the list. The list is then composed with every integer number between these values. Because of this, the first value must be smaller then the last

```
[1..5] // [1,2,3,4,5]
[3..7] // [3,4,5,6,7]
[5..3] // Invalid
```

The second variants specifies the first, second and last value for the list. The increment is the difference between the second and first value of the list, which can even be negative.

The increment is then added to each element until the largest possible value which is smaller than or equal to the last value. If the increment is negative, the list stops at the smallest possible value which is larger than or equal to the last value.

```
[1,3..10] // [1,3,5,7,9]
[5,4..1] // [5,4,3,2,1]
[5,3..0] // [5,3,1]
```

Comprehensions List comprehensions are a simple way to transform every value in a list, creating a new list.

In the example below, `ls` is a list containing every number from 1 to 10, inclusive. Using a list comprehension, `new` is a list containing every number from 2 to 11, since the code `x+1` is executed for every value in `ls`.

```
let ls = [1..10];
let new = [x+1 for x in ls];
```

2.2.2 Tuples

Tuples group multiple values, possibly of different types, into a single compound value. The minimum size of a tuple is 2, but there is no limit on its maximum size.

Tuples are specified inside parenthesis, with each of its values separated by commas.

```
(1, ``hello'')
(true, `c', 43)
```

Tuples are immutable, which means they cannot be changed once they are created. There is no way to add or remove elements from a tuple.

To access a specific field of a tuple, the projection operation is used. Tuples are 0-indexed, so projection is done by specifying the index of the field.

```
#0 (1, ``hello'')
#2 (true, `c', 43)
#3 ('c', false) // Invalid code
```

Tuples are extremely useful as return values for functions that must convey more than one piece of information. Since every function can only return one value, tuples can be used to group the different values that the function must return.

2.2.3 Records

Records are, like tuples, groupings of multiple values of possibly different types. Unlike tuples, however, each value has its unique label. The smallest size for a record is 1, but there is no limit on its maximum size.

To construct a record, each value must be preceded by a label and a colon. Each label-value pair is separated by a comma, and the whole record is enclosed in curly brackets ({ }).

```
{name: ``Martha'', age: 32}
{day: 1, month: 1, year: 2000}
```

Records are, like tuples, immutable. To access a specific field of a record, one must use projection. There is no inherent order in the values of a record, so the label must be used for projection.

```
#name {name: ``Martha'', age: 32}
#month {day: 1, month: 1, year: 2000}
#name  {day: 1, month: 1, year: 2000} // Invalid code
```

Records are also useful as return values for functions. The advantage they have over tuples is that every field has its name. This removes ambiguities for values of the same type, such as in the case of a date with 3 integer values.

2.3 Identifiers

Identifiers are used to name constants (in let declarations), functions and function arguments. When an identifier is expected, the identifier is defined as the longest possible sequence of valid characters. Any Unicode character is considered valid, with the exception of the following:

.	,	;	:	!	@	&
+	-	/	*	<	=	>
(	)	{	}	[	]	|
%	\	’	”	\n	\r	\t
_	‘					

Numerical digits are not allowed at the start of an identifier, but they can be used in any other position.

Furthermore, *V* has some reserved names that cannot be used by any identifier. They are the following:

let	true	false	if	then	else
rec	nil	raise	when	match	with
for	in	import	infix	infixl	infixr

2.4 Patterns

Patterns are rules for deconstructing values and binding their parts to identifiers. They can be used in constant declarations and function parameters, simplifying the extraction of data from compound values.

Pattern	**Examples**	**Comments**
Identifier	x, y, z	Matches any value and binds to the identifier
Number	1, 3	Matches and ignores the number
Boolean	true, false	Matches and ignores the boolean
Character	‘c’, ‘f’	Matches and ignores the character
Identifier	x, y, z	Matches any value and binds to the identifier
Wildcard	_	Matches and ignores any value
Tuple	(x, _, y)	Matches tuples with corresponding size
Record	{a: _, b: (x,y), c: x}	Matches records with corresponding labels
Nil	nil, []	Matches the empty list
Cons	x :: y	Matches the head and tail of a non-empty list
List	[x, y, z]	Matches lists of corresponding size

Compound patterns, such as `Tuple`, `Record` and `List`, are composed of other patterns separated by commas. All patterns can have optional type annotations added to explicitly declare their types.

One thing to notice is that all patterns related to lists (`List`, `Cons` and `Nil`) and values (numbers, characters) can fail. If an attempt to match a pattern fails (e.g matching a non-empty list with `Nil`), the expression will raise an exception.

2.5 Constants

Constants are associations of identifiers to values. The value associated to a particular identifier cannot be changed after it is declared.

The keyword `let` is used to start a constant declaration, and a semicolon ends it. After the `let` keyword, any pattern can be used.

Below are examples of constant declarations:

```
let name = ``Steve'';
let age: Int = 32;
let (x: Int, y) = (4, true);
```

2.6 Type Annotations

Type annotations are used to explictly state the type of a constant, function argument or function return value. They are not necessary for most programs, since the interpreter can infer the type of any expression.

Sometimes, the programmer may want to create artificial constraints on a function argument, and type annotations allow this.

The table below shows every type that can be specified in type annotations. These types align with the types available in the *V* language, since every type can be used in a type annotation.

Type	**Example Values**	**Comments**
Int	1, 0, -3	
Bool	`true`, `false`	
Char	'c', ' '	
String	"abc", ""	This is syntactic sugar for `[Char]`
[*Type*]	[1, 2, 3], `nil`	List Type
(*Type*, ... *Type*)	(1, true, 'a')	Tuple Type
{id: *Type*, ... id: *Type*}	{a: 3, b: false}	Record Type
Type -> *Type*		Function Type (see 2.10.5)

2.7 Conditionals

V provides a conditional expression (`if ... then ... else`) to control the flow of a program. This expression tests a condition and, if its value is `true`, executes the first branch (known as the `then` clause). If the condition is `false`, the expression executes the second branch (the `else` clause).

```
if b then
  1+3 // Will execute if b is true
else
  2 // Will execute if b is false
```

The only accepted type for the condition of a conditional is Boolean. All types are accepted in the `then` and `else` branch, but they must be of the same type.

```
// This conditional is invalid code, since 4 and "hello" are of different type
if true then
  4
else
  "hello"
```

Unlike imperative languages, every conditional in *V* must specify both branches. This ensures that the conditional will always return a value.

It is possible to chain multiple conditionals together.

```
if grade > 10 then
  "The grade cannot be higher than 10"
else if grade < 0
  "The grade cannot be lower than 0"
else
  "The grade is valid"
```

2.8 Match Expressions

Match expressions are another way to control the flow of a program based on comparison with a list of patterns. Any number (greater or equal to 1) of patterns can be specified, and each one has a corresponding result expression.

```
match value with
| pattern1 -> result-expression1
| pattern2 when condition -> result-expression2
...
```

Each pattern is tested from top to bottom, stopping the comparison as soon as a valid match is found. When this happens, the corresponding result expression is evaluated and returned.

It is also possible to specify an aditional condition that must be satisfied for a pattern to be accepted. This condition can use any identifiers declared in its corresponding pattern, and it is not evaluated unless the pattern pattern returns a correct match.

2.9 Operators

V contains a number of infix binary operators to manipulate data.

Along with them, there is only one prefix unary operator, the negation operator. This operator is handled differently from a function application, both in its priority and its associativity.

2.9.1 Priority

Every operator is ordered within a priority system, in which operators at a higher priority level are evaluated first. The levels are ordered in numerical order (i.e. priority 9 is the highest level). For different operators at the same priority level, the evaluation is always done from left to right.

2.9.2 Associativity

Some operators can be composed several times in a row, such as addition or function application. For these operators, it is necessary to define how they are interpreted to return the desired value. There are 2 possible associativities that an operator can have:

- Left-associative
  `((a + b) + c) + d`
- Right-associative
  `a + (b + (c + d))`

2.9.3 Table of Operators

Below is a summary of every operator avaiable in the language, along with a small description and their associativities (if any). The table is ordered by decreasing priority level (the first operator has the highest priority).

Priority	Operator	Meaning	Associativity
10	`f x`	Function Application	Left
9	`f . g`	Function Composition	Right
	`x !! y`	List Indexing	Left
8	`x * y`	Multiplication	Left
	`x / y`	Division	Left
	`x % y`	Remainder	Left
7	`x + y`	Addition	Left
	`x - y`	Subtraction	Left
	`- x`	Unary Negation	None
6	`x :: y`	List Construction	Right
5	`x @ y`	List Concatenation	Right
4	`x == y`	Equals	None
	`x != y`	Not Equals	None
	`x > y`	Greater Than	None
	`x >= y`	Greater Than Or Equal	None
	`x < y`	Less Than	None
	`x <= y`	Less Than Or Equal	None
3	`x && y`	Logical AND	Right
2	`x || y`	Logical OR	Right
1	`x $ y`	Function Application	Right

2.9.4 Operators as Functions

It is possible to use any operator as a function by wrapping it in parenthesis. The left-hand operand becomes the first argument of the function, and the right-hand operand becomes the second argument.

This is useful mainly when passing operators as arguments to functions.

```
// Both expressions are equivalent
zipWith (\x y -> x + y) [1,2,3] [3,2,1]
zipWith (+) [1,2,3] [3,2,1]

// Adds 2 to every element in the list
map ((+) 2) [1,2,3]
```

As is shown in the last example, it is possible to provide the left-hand operand to obtain a partially applied function. If one wishes to provide the right-hand operand instead, it is possible to use the `flip` function, which changes the order of a function that takes two parameters.

```
// Divides 2 by every value in the list
map ((/) 2) [1,2,3]

// Divides every value in the list by 2
map (flip (/) 2) [1,2,3]
```

2.9.5 Functions as Operators

Wrapping a function name in backticks (‘) will turn it into an infix binary operator. The first parameter of the function will become the left-hand operand, while the second parameter will become the right-hand operand.

```
4 `add` 5
add 4 5
```

It is possible to use this with functions that take more than 2 parameters, but then it becomes necessary to use parenthesis to pass the remaning parameters. This greatly reduces the readability of the code, and is therefore not encouraged.

2.9.6 Defining new Operators

Is is possible to define new operators to be used like regular operators. The syntax for this is the same as creating a new function, but the operator must be enclosed in parenthesis.

```
let (%+) x y = x % y + 1;

5 %+ 4 // 2
```

When declaring an operator, it is possible to also define its associativity and priority. This is done by using the keywords `infixl` (left associative), `infixr` (right associative) and `infix` (non-associative), followed by a number from 1 to 9.

```
let infixl 1 ($) f x = f x;
let f x = x + 2;

f $ 4 // 6
```

If the associativity and priority information is not provided, the operator will have priority 1 and be left associative.

The following are the list of characters allowed for operators.

?	!	%	&	*	+
-	.	/	<	=	>
@	^	|	~		

2.10 Functions

There are 4 types of functions that a programmer can declare:

1. Named functions
2. Recursive Named functions
3. Lambdas (unnamed functions)
4. Recursive Lambdas

2.10.1 Named Functions

These are functions that have a name by which they can be called after their definition. After the name, the programmer must specify one or more parameters, which can be any pattern. If an explicitly typed pattern is used, it must be enclosed in parenthesis. After every argument, the programmer can specify the return type of the function.

The body of a function can use any parameter declared in its definition to compute a return value. Since every expression in the language returns a value, any valid expression is accepted as the body of a function. The only constraint is that, if the definition specifies a return type, the value must be of that type.

Below are three examples of named functions:

```
let add x y =
  x + y
;

let duplicate (x: Int): Int =
  x * 2
;

let addTuple (x, y) =
  x + y
;
```

2.10.2 Recursive Named Functions

These functions differ from regular named functions by the fact that they can be called from within their own body. This means that the function can be called recursively, iterating over a certain value (or values). To indicate that a function is recursive, the keyword `rec` is added before its name. Below are two examples of recursive named functions:

```
let rec count ls =
  if empty? ls then
    0
  else
    1 + count (tail ls)
;

let rec factorial (x: Int): Int =
  if x == 0 then
    1
  else
    x * factorial (x - 1)
;
```

Here, both functions perform a test that determines whether the end condition is met. If the end condition is met, the function returns a simple value. If the end condition is not met, the function recursively calls itself with a modified value, continuing the iteration.

In the case of the `count` function, the recursion terminates when the input is an empty list. For the `factorial` function, an input equal to 0 terminates the recursion.

2.10.3 Lambdas

These are simple unnamed functions with a compact syntax that allows them to be written in a single line most of the time. This is useful mostly when passing lambdas as arguments to other functions, since they do not require creating a full named declaration.

The general syntax of a lambda is as follows:

```
\param1 param2 ... -> body
```

A backslash (\) indicates the start of a lambda, followed immediately by its parameters. Like in named functions, these can be any valid pattern. Unlike named functions, however, the return type of a lambda is never specified.

After the parameters, an arrow (-¿) indicates the start of the function body, which extends as far to the right as possible. Because of this, lambdas are usually enclosed in parenthesis to limit their scope.

Below are the same examples shown in the named functions section, but defined using lambda expressions. Notice that, without the use of parenthesis to enclose each lambda, the first function would try to include everything inside its body, resulting in a parsing error.

```
\\ add
(\x y -> x + y)

\\ duplicate
(\(x: Int) -> x * 2)

\\ AddTuple
(\(x, y) -> x + y)
```

2.10.4 Recursive Lambdas

Just like there is a recursive variant of named functions, there is a recursive variant of lambdas. These are compact expressions to define recursive functions. Like regular lambdas, they are used mostly to be passed as arguments to other functions, and are usually enclosed in parenthesis.

Unlike for regular lambdas, it is necessary to specify a name for a recursive lambda. Without a name, it would be impossible to call itself within its body. It is important to realize that, unlike with recursive named functions, this name is limited in scope to the inside of the lambda definition.

```
(rec fac x -> if x == 0 then 1 else x * fac (x - 1))
```

```
fac 4 // This is invalid code
```

With the example above, we see that the programmer tried to call a recursive lambda outside its definition. The offending code is outside the scope in which `fac` is available, resulting in invalid code.

2.10.5 Function Type

Every function has a type consisting of its parameter types and return type. Every parameter type is separated by an arrow (->), and the return type is also separated by a single arrow from the parameter types.

The syntax for a function type is as follows:

```
param1 -> param2 -> ... -> return
```

If one of the parameters is itself a function, it is possible to use parenthesis to indicate this.

The following function type defines a function that takes two parameters. The first is a function of type `Int -> Int`. The second parameter is an `Int`, and the return type is also `Int`.

```
(Int -> Int) -> Int -> Int
```

If the parenthesis were ommited, the type would describe a function that takes 3 parameters of type `Int`.

2.11 Partial Application and Currying

Technically, every function in *V* takes only one parameter. When a function is defined as having multiple parameters, it is actually a curried function.

As an example, take the following function, which returns the largest of two numbers:

```
let max x y =
  if x > y then
    x
  else
    y
;
```

This appears to be a function that takes two integers and returns an integer. In reality, `max` is a function that takes one integer and returns another function. This returned function takes one integer as a parameter and returns another integer.

This allows what is called *partial application*, which is when a function is called with too few arguments. This creates a function that "fixes" the applied arguments and returns a function that takes the remaining arguments.

Using the example above, we can write `max 5` to create a new function that takes only one argument. This function will then return the largest between its argument and the number 5.

It can then be bound to a name, just like any other function, and used elsewhere. This is also useful for quickly creating new functions with fixed data or to be passed as arguments.

```
let max5 = max 5;

max5 3 // Returns 5
max5 10 // Returns 10
```

String Conversions There are available functions to convert integers and booleans into and from strings. There are no included functions to convert compound types, but it is possible to create custom ones for each use case.

To convert strings to integers, the function is `parseInt`.

To convert integers to strings, the function is `printInt`.

To convert strings to booleans, the function is `parseBool`.

To convert booleans to strings, the function is `printBool`.

2.12 Comments

Comments are text that is ignored by the interpreter. They can be used to add notes or reminders for yourself or anyone that reads the source code.

Currently, only single line comments are avaiable. They begin with two forward-slashes (//) and continue until the end of the current line

```
// This is a comment on its own line.
3 + 4 // This line has code and a comment.
```

2.13 Libraries

Libraries are collections of constant and function declarations designed to be reused in multiple programs. These files can then be compiled to increase loading times when interpreting programs, or be loaded as parseable text files on their own.

To import a library in a program, the following syntax is used:

```
import "library"
```

The name of the library must be a string indicating the path of the library file. This path can either be relative to the program that is being executed or absolute. If a file extension is not provided for the file, it is assumed to be *vl*, which is the default extension used for compiled *V* libraries, or *v*, which is the extension for source code files in *V*.

Libraries can be imported anywhere in a program, and their functions will have their scope limited to wherever they were imported.

In the example below, we have a library with a single function `double`. This library is then imported inside a function in a program. Because the library was imported inside the scope of the function body, none of its functions can be called outside of it.

```
// math.vl
let double x = x * 2;

//-----------------
// Program.v
let quadruple x =
  import "math"
  double (double x) // Valid
;

double 4 // Invalid
```

3 Standard Library

The Standard Library, called `stdlib`, is always imported in every *V* program. It provides basic functions for a number of use cases, ranging from numerical operations to function manipulation.

Some basic language features, such as list comprehensions and ranges, depend on the existence of the `stdlib`. This means that, while it is possible to create programs without importing the `stdlib`, doing so will most likely break any existing program.

3.1 Operations on Basic Values

3.1.1 Operations on Numbers

The 4 basic operations (addition, subtraction, multiplication and division) are built into the language. Other operations must be defined in terms of these.

One important thing to note is that the unary negation operator (`-`) is tightly coupled with the `negate` function defined in the `stdlib`. While the operator is defined inside the language, it depends on the presence of the `stdlib` to funciton.

remainder :: *Int* → *Int* → *Int*

Integer remainder, satisfying:

```
(x / y) * y + (remainder x y) = x
```

(%) **::** *Int* → *Int* → *Int* | left-associative, priority 8 |

Infix version of `remainder`

negate :: *Int* → *Int*

Unary negation, satisfying:

```
x + (negate x) = 0
```

abs :: *Int* → *Int*

Absolute value

3.1.2 Operations on Booleans

Below are all the operations on booleans defined in the Standard Library. Missing from these are the basic AND (`&&`) and OR (`||`) operations. This is because they require short-circuit evalaution, and must therefore be built into the language.

not :: *Bool* → *Bool*

Boolean “not”

```
not True = False
not False = True
```

xor :: *Bool* → *Bool* → *Bool*

Boolean “xor”

```
xor True True = False
xor True False = True
xor True False = True
xor False False = False
```

3.1.3 Operations on Functions

Basic manipulation of functions and application. Most of the usefulness of these functions come from their infix versions. They allow more compact and easier to read code to be written, mainly reducing the need for parentheses.

flip :: $(a \to b \to c) \to b \to a \to c$

`flip f` takes its first two arguments in reverse order of `f`.

```
flip f x y = f y x
```

apply :: $(a \to b) \to a \to b$

This function simply applies its second argument to its first. While this seems redundant (after all, `apply f x` is the same as `f x`), it can be used higher order situations.

($) **::** $(a \to b) \to a \to b$ | right-associative, priority 1 |

Infix version of `apply`. While it has the same funcionality as normal function application, it is right-associative with the lowest possible priority.

In some situations, this allows parentheses to be omitted.

```
f $ g $ h x = f (g (h x))
```

compose :: $(b \to c) \to (a \to b) \to a \to c$

Function composition. Applies the third argument to the second one, applying the resulting value to the first argument.

```
compose f g x = f (g x)
```

(.) **::** $(b \to c) \to (a \to b) \to a \to c$ | right-associative, priority 9 |

Infix version of `compose`.
Can be used with `$` to reduce the number of parentheses needed.

```
f . g . h $ x = f (g (h x))
```

3.1.4 Operations on Tuples

The stdlib also provides basic functions for manipulating tuples with 2 components. For larger tuples, it is necessary to create custom functions.

fst :: $(a,b) \to a$

Returns the first component of a pair.

snd :: $(a,b) \to b$

Returns the second component of a pair.

swap :: $(a,b) \to (b,a)$

Swap the components of a pair.

3.2 Operations on Lists

3.2.1 Basic Operations

Basic functions to aid in using lists.

head :: $[a] \to a$

Returns the first element of a list, which must have at least one element.

last :: $[a] \to a$

Returns the last element of a list, which must have at least one element.

tail :: $[a] \to [a]$

Removes the first element of a list, which must have at least one element.

init :: $[a] \to [a]$

Removes the last element of a list, which must have at least one element.

tail :: $[a] \to [a]$

Removes the first element of a list, which must have at least one element.

empty? :: $[a] \to Bool$

Returns `True` if the list is empty, and `False` otherwise.

length :: $[a] \to Int$

Returns the number of elements in the list.

append :: $a \to [a] \to [a]$

Adds an element to the end of a list.

concat :: $[a] \to [a] \to [a]$

Appends two lists, maintaning order.

(@) **::** $[a] \to [a] \to [a]$ | right-associative, priority 5 |

Infix version of `concat`.

3.2.2 Generation Operations

These operations create lists based on input values.

range :: $Int \to Int \to Int \to [Int]$

`range start finish increment` generates a list of the form
[`start`, `start` + `increment`, `start` + 2 * `increment`, ..., `n`], where
`increment` > 0 $\implies$ `n` ≤ `finish`
`increment` < 0 $\implies$ `n` ≥ `finish`

3.2.3 Transformation Operations

These operations transform a list, altering its elements, their order, or both.

reverse :: $[a] \rightarrow [a]$

Returns the elements of the input in reverse order.

map :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`map f ls` returns a list by applying the function `f` to each element of the list `ls`.

3.2.4 Reduction Operations

These operations take a list and reduce it to a simple value.

fold :: $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

`fold f acc ls` reduces the list using the function `f`, applying it to an accumulator (`acc`) and each element of the list, from left to right.

reduce :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$

The same as `fold`, but using the first element of the list as the `acc`

all :: $(a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

Checks whether all elements of a list satisfy a predicate. An empty list returns true.

any :: $(a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$

Checks whether any elements of a list satisfy a predicate. An empty list returns false.

maximum :: $Orderable\ a$ => $[a] \rightarrow a$

Returns the largest element of the list.

minimum :: $Orderable\ a$ => $[a] \rightarrow a$

Returns the smallest element of the list.

3.2.5 Sublist Operations

These operations return smaller segments of an existing list.

take :: $Int \to [a] \to [a]$

`take n ls` returns the first `n` elements of `ls`.

drop :: $Int \to [a] \to [a]$

`drop n ls` returns the list resulting from removing the first `n` elements of `ls`.

takeWhile :: $(a \to Bool) \to [a] \to [a]$

`takeWhile p ls` returns the longest prefix of `ls` such that every element satisfies `p`.

dropWhile :: $(a \to Bool) \to [a] \to [a]$

`dropWhile p ls` returns the suffix that remains after `takeWhile p ls`.

sublist :: $Int \to Int \to [a] \to [a]$

`sublist start length ls` drops the first `start` elements of `ls`, and then takes the first `length` elements of the resulting list.

3.2.6 Search Operations

These operations search for specific elements in a list.

exists :: $Equatable\ a => a \to [a] \to Bool$

Tests whether the given element exists in the list.

filter :: $(a \to Bool) \to [a] \to [a]$

`filter p ls` returns a sublist of `ls` such that every element satisfies `p`.

3.2.7 Indexing Operations

Manipulate a list through the index of its elements

indexOf :: *Equatable a => a* $\to$ [*a*] $\to$ *Int*

`indexOf t ls` returns the index of the first occurrence of `t` in `ls`. If the element does not occur, returns `-1`.

nth :: *Int* $\to$ [*a*] $\to$ *a*

`nth n ls` returns the element of `ls` at position `n`. If `n` is negative or larger than `length ls`, an exception is raised.

(!!) **::** [*a*] $\to$ *Int* $\to$ *a* | left-associative, priority 9 |

The infix version of `nth`. It receives its operands in reverse order, allowing for expressions in the form `ls !! n`.

3.2.8 Sorting Operations

Sort lists.

sort :: *Orderable a =>* [*a*] $\to$ [*a*]

Sorts a list in ascending order.

3.2.9 Zipping Operations

Operations that deal with tuples and lists.

zip :: [*a*] $\to$ [*b*] $\to$ [(a, b)]

Takes two lists and returns a list composed of corresponding pairs. It the lists have different lengths, elements of the larger one are discarded.

zipWith :: $(a \to b \to c) \to [a] \to [b] \to [c]$

Takes two lists and a function, returning a list composed of the result of applying the function to corresponding elements in each list. It the lists have different lengths, elements of the larger one are discarded.

unzip :: $[(a, b)] \to ([a], [b])$

Takes a list of pairs, returning a pair of lists, each containing the corresponding components of the original list.

3.3 String Conversion Operations

4 Changelog

v0.2

May 1, 2017

Additions

- **Match Expressions** Abstract Syntax Operational Semantics Type System Concrete Syntax

  This is a structure to control the flow of a program by attempting to match a value against a list of patterns.

- **Prefix Notation for Operators** Concrete Syntax

  Wrapping any infix operator in parenthesis turns it into a function that takes two parameters.

- **Infix Notation for Functions** Concrete Syntax

  Wrapping any function name in backticks (‘) turns it into a binary operator.

- **Creating new Operators** Concrete Syntax

  When declaring a function, wrapping a name in parenthesis makes it a operator. Only a small number of characters are allowed for operator names, and it is possible to define the associativity and priority of the newly created operator (if this information is ommited, the default values are left associative and priority 9).

Changes

- **List Operations** Abstract Syntax Operational Semantics Type System

  The list operations `hd`, `tl` and `isempty` have been removed from the language defition. They have been added to the stdlib, and use pattern matching to recreate their funcionality.

- **Non-Strict Semantics** Operational Semantics

  Functions can now be non-strict, and compound values (lists, tuples, records) can contain undefined (`raise`) components.

- **Boolean Operations** Abstract Syntax Operational Semantics Type System

  The boolean operators $\wedge$ (AND) and $\vee$ (OR) have been removed from the definition and added to the stdlib.

Removals

- **Input and Output** Abstract Syntax Operational Semantics Type System Concrete Syntax

  Removed the input and output expressions. This was done because they are not compatible with the functional nature of V. IO will (probably) be added again in the future, but for now it must be removed.

- **Sequence Operator, Skip and Unit** Abstract Syntax Operational Semantics Type System Concrete Syntax

  Since these are not used by any expressions, they have been removed.

- **Try** Abstract Syntax Operational Semantics Type System Concrete Syntax

  Exceptions can no longer be handled. This was done because of the change to non-strict semantics.