1 Language Guide

1.1 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’).

1.2 Compound Values

1.2.1 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 **??**) 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];
```

1.2.2 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, it must be deconstructed using a pattern. This can be done either by using a `let` expression or by pattern matching in a function definition.

```
let (x, y) = (1, ``hello'')
(\&&(x, y, z) -> x) (true, `c', 43)
```

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.

1.2.3 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}
```

To get the value of a single field in a record, one can use the `get` function, passing the name of the field and the record. As can be seen below, the name of the field must be prefixed by the `#` character.

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

Unlike tuples, it is possible to alter a single field in a record. This is done by using the `set` function, which takes the name of the field (with the `#` character), the value to be set and the record.

It is also possible to modify a value of a field by using the `modify` function, which applies a function given as its parameter to the existing value.

It is important to note that these function do not change the original record, but instead returns a new, modified record.

```
set 3 #age {name: ``Martha'', age: 32}
// Returns {name: ``Martha'', age: 3}
set 8 #month {day: 1, month: 1, year: 2000}
// Returns {day: 1, month: 8, year: 2000}

modify #age (\x -> x * 2) {name: ``Martha'', age: 32}
// Returns {name: ``Martha'', age: 64}
```

1.3 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

1.4 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
Partial Record	{a: _, b: (x,y), c: x, ...}	Matches records with at least the specified 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.

1.5 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);
```

1.6 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 1.10.5)

1.7 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"
```

1.8 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.

1.9 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.

1.9.1 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.

1.9.2 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))`

1.9.3 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

1.9.4 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]
```

1.9.5 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.

1.9.6 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.

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

1.10 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

1.10.1 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
;
```

1.10.2 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.

1.10.3 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)
```

1.10.4 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.

1.10.5 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`.

1.11 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`.

1.12 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.
```

1.13 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
```