VIVA

User’s Guide

vanb’s Input Verification Assistant v2.0

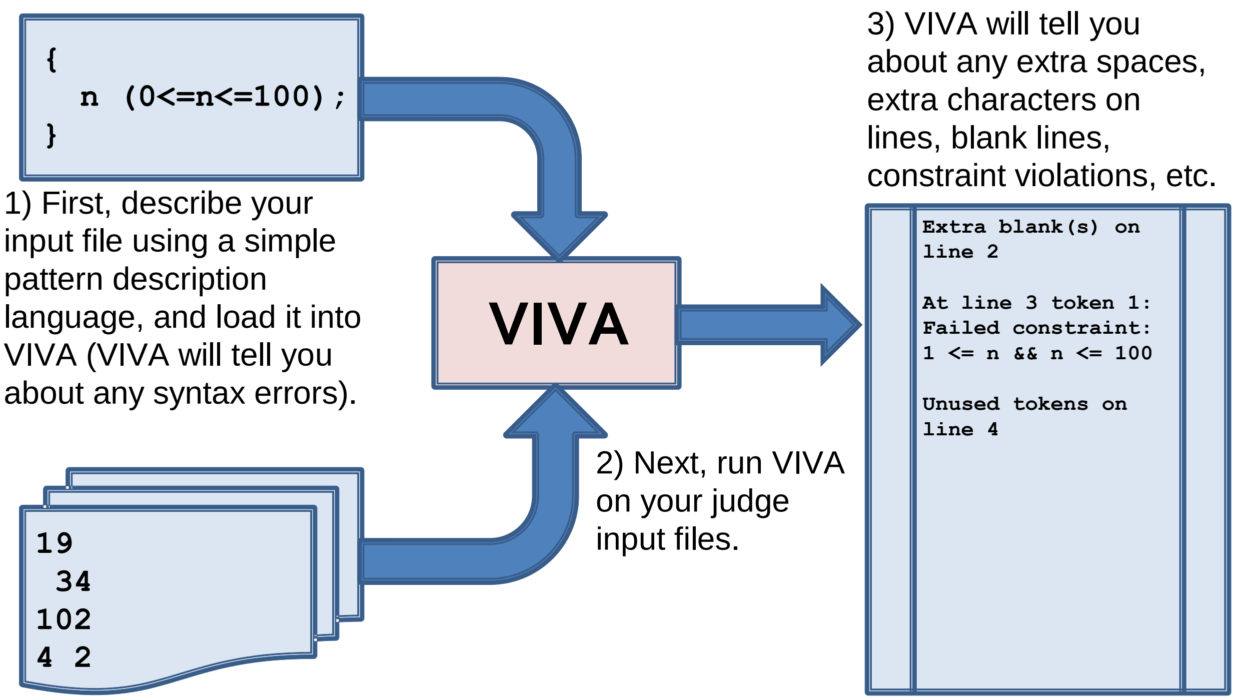
VIVA: vanb's Input Verification Assistant User's Guide

Table of Contents

Introduction

In running any computer programming contest, ensuring the accuracy of judge data can be a challenge. The judges may write programs to solve the problems, which will reveal some data errors, but still, some simple issues can go undetected, such as blank lines, extra spaces, values outside of specified constraints, and so on. Many judges are former competitors who write very robust programs, which are often immune to troubles caused by such simple formatting and constraint errors. However, these kinds of issues can cause problems with competitor's code during the contest, causing, at best, extra effort and delays by the judges during the contest, and at worst, incorrect judge responses.

VIVA is vanb's Input Verification Assistant. It takes a description of an input specification in a simple language, and can then test input files to ensure that they conform to the input specification.


3) VIVA will tell you about any extra spaces, extra characters on lines, blank lines, constraint violations, etc.
1) First, describe your input file using a simple pattern description language, and load it into VIVA (VIVA will tell you about any syntax errors).
VIVA
2) Next, run VIVA on your judge input files.

```
{
   n  (0<=n<=100);
}
```

```
19
 34
102
4  2
```

```
Extra blank(s) on
line 2

At line 3 token 1:
Failed constraint:
1 <= n && n <= 100

Unused tokens on
line 4
```

Disclaimer

VIVA is intended as a tool to aid in the process of verifying judge input data, but not to be the only step in that process. It is meant to augment your verification process, not to replace it or reduce it. VIVA is only as good as the patterns written for it. In addition, there are often complex relationships in data that cannot be represented in VIVA. The author accepts no responsibility for errant judge data that passes a verification process that includes VIVA.

Running VIVA

There are currently three ways to use VIVA: from the command line, embedded in a program, and via a VIVA GUI.

From the Command Line

VIVA is a Java application, so it must be run from the Java runtime environment.

```
java –jar viva.jar patternfile [inputfile]*
```

If the input files are omitted, VIVA will only parse the pattern file. If the pattern file is omitted, VIVA will display a helpful message.

When run from the command line, VIVA produces an error code. This code is `-2` for bad usage (i.e. no parameters), or `-1` if the pattern file fails to parse. Otherwise, VIVA will return the number of files which failed validation (`0` if all files passed).

Embedding VIVA in a Program

VIVA is a Java application, so it can easily be embedded in a Java program. VIVA is entirely contained in the file **`viva.jar`**, which should be in the Java classpath.

First, create an instance of VIVA, using the constructor:

```
public VIVA()
```

Normally, VIVA will send its output to **`System.out`**, but it can be configured to send the output to a stream of your choice. If you wish to use an output stream other than **`System.out`**, use this method:

```
public void setOutputStream( PrintStream ps )
```

It is also possible to add your own functions to VIVA. It's usually not necessary, since VIVA's included set of functions is fairly complete. However, if you're going to add functions, it should be done here, before parsing a pattern. This is an advanced topic which will be covered in a later chapter. For completeness, the methods are listed here.

```
public void addFunction( ScalarFunction function )
public void addFunction( VectorFunction function )
```

The next two methods form the core of VIVA's functionality: parsing a pattern, and testing an input file. They may be repeated. The last pattern parsed will be applied to all subsequent input files.

This method sets the active pattern in VIVA, and, as a consequence, parses the pattern.

```
public void setPattern( InputStream stream )
```

The last method tests an input file. It will return **`true`** if the input file passed, **`false`** if it failed. Diagnostic messages will be written to **`System.out`** by default, or whatever output stream has been specified.

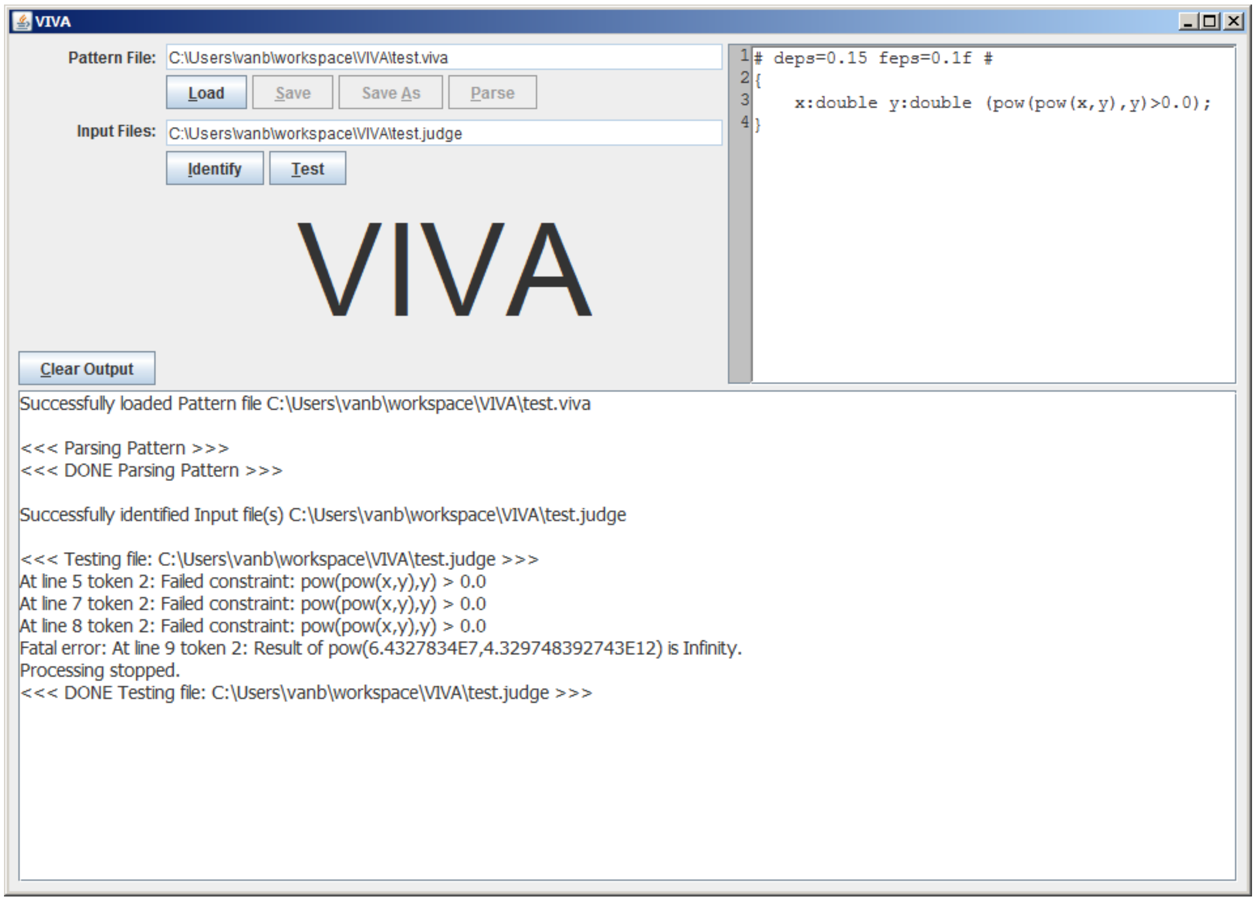
```
public boolean testInputFile( String filename )
```

VIVA GUI

VIVA provides a very simple GUI. It also must be run with the Java runtime:

```
java -jar vivagui.jar
```

The interface looks like this:



Pattern File: C:\Users\vanb\workspace\VIVA\test.viva

Input Files: C:\Users\vanb\workspace\VIVA\test.judge

```
# deps=0.15 feps=0.1f #
{
    x:double y:double (pow(pow(x,y),y)>0.0);
}
```

```
Successfully loaded Pattern file C:\Users\vanb\workspace\VIVA\test.viva

<<< Parsing Pattern >>>
<<< DONE Parsing Pattern >>>

Successfully identified Input file(s) C:\Users\vanb\workspace\VIVA\test.judge

<<< Testing file: C:\Users\vanb\workspace\VIVA\test.judge >>>
At line 5 token 2: Failed constraint: pow(pow(x,y),y) > 0.0
At line 7 token 2: Failed constraint: pow(pow(x,y),y) > 0.0
At line 8 token 2: Failed constraint: pow(pow(x,y),y) > 0.0
Fatal error: At line 9 token 2: Result of pow(6.4327834E7,4.329748392743E12) is Infinity.
Processing stopped.
<<< DONE Testing file: C:\Users\vanb\workspace\VIVA\test.judge >>>
```

The interface consists of four text fields and seven buttons.

The **Pattern File** and **Input Files** fields display the names of the pattern file and input file (or files) currently selected. These text fields are navigable, but they are not editable. Use the **Load** and **Save As** buttons to select a pattern file, and use the **Identify** button to identify input file(s) for testing.

The text box with line numbers at the top right is the Pattern Editor. Here, you can create and edit patterns.

The text box which consumes the lower half of the window is an Output box. This box will display all of the messages from VIVA as a result of parsing patterns or testing input files. Also, this box will show messages from the GUI confirming actions, or reporting errors. It is also not editable.

Button	Action	Active / Inactive
Load	Pops up a Pattern File Chooser dialog to select a Pattern File. If a file is chosen, the contents are loaded into the Pattern Editor and immediately parsed.	This button is always active.
Save	Saves the contents of the Pattern Editor to whatever file was last specified in the Pattern File Chooser through a **Load** or **Save As**.	This button is active whenever the text in the Pattern Editor is not consistent with the specified Pattern File. It activates with any change to the pattern in the Pattern Editor, and deactivates with a successful **Load**, **Save** or **Save As**.
Save As	Pops up a Pattern File Chooser dialog to select a Pattern File. If a file is chosen, the contents of the Pattern Editor are written to that file.	This button is active whenever the text in the Pattern Editor is not consistent with the specified Pattern File. It activates with any change to the pattern in the Pattern Editor, and deactivates with a successful **Load**, **Save** or **Save As**.
Parse	Parses the pattern in the Pattern Editor. If successful, this becomes the pattern that will be used to test input files.	This button is active whenever the text in the Pattern Editor is not consistent with the parsed pattern being used for testing. It activates with any change in the Pattern Editor, and deactivates on any successful parsing.
Identify	Pops up an Input File Chooser to identify input files for testing. This Chooser allows multiple files to be chosen.	This button is always active.
Test	Tests the identified input files by running VIVA with the parsed pattern on them. The results are displayed in the Output box.	This button is active whenever the text in the Pattern Editor has been successfully parsed. It deactivates with any change in the Pattern Editor.
Clear Output	Erases all text in the Output box.	This button is always active.

Patterns

The pattern specification is the heart of VIVA. VIVA offers a simple yet powerful syntax which covers most of the patterns and constraints typically seen in judge input files.

Simple Patterns

The simplest pattern is just a variable, with constraints. It looks like this:

```
variable ( constraints )
```

The constraints are optional. The variable name must conform to Java identifier specifications, with one exception: Dollar signs (‘**$**’) are NOT allowed in variable names.

For example, these are all legitimate simple patterns:

```
x
n (n<3)
k (0<=k<=10, k!=5)
```

Without a type specification (described in the next section), the variables are assumed to be integers. Multiple constraints may be given, separated by commas (‘**,**’). Comparators can be chained; the constraint **0<=k<=10** is parsed as **0<=k && k<=10**.

Multiple patterns can be put on the same line, terminated with a semicolon. This means that all of the matching values in the input file must appear on the same line, with a single space between them, and with no leading or trailing blanks. Constraints can use any previously mentioned variable within scope. For example:

```
x (0<=x<=100) y (0<=y<=100);
m n (0<n<=20, n<m<=30);
p q (0<p<q<80) r (0<r<p*q);
```

Other Data Types

VIVA supports the following data types: **int**, **integer**, **long**, **float**, **double**, and **string**. The first two, **int** and **integer**, are synonyms. If no type is specified, **integer** is assumed. Specify a type by following the variable name with “**:*type***” For example:

```
x y:double z:string;
```

Means: Read three values: an integer, followed by a double, followed by a string. Input strings can NOT contain spaces. For example, if an input line was:

```
1 2 3
```

The **1** would be interpreted as an **integer**, read into **x**.

The **2** would be interpreted as a **double**, read into **y**.

The **3** would be interpreted as a **string**, read into **z**.

Constraint Constants

The constants in constraints follow Java conventions. This table lists some examples:

Constant	Type
`19`	integer
`10000000000L`	long (lower case '**`l`**' works, too)
`10000000000`	Parser Error (too big for integer, no '**`L`**' for long)
`0xAB`	integer (in hex)
`0712`	integer (in octal)
`0281`	Parser Error (8 is not an octal digit)
`1.2`	double
`1E+10`	double (lower case '**`e`**' works, too)
`19F`	float (lower case '**`f`**' works, too)
`"Hi There"`	string
`"19"`	string
`'A'`	integer, value is 65

VIVA does not support a **`char`** type, but it can accept **`char`** constants. They are converted to their integer ASCII equivalents.

Constraint Operators

Constraints are expressions which evaluate to **`true`** or **`false`**. VIVA offers some operators outside of the standard Java/C++ operators, and the precedence is different. The following table describes VIVA's operators, and their precedence, from low to high.

Precedence Level	Operator	Meaning
Logical OR	`||`	Logical OR
	`^^`	Logical XOR
Logical AND	`&&`	Logical AND
Comparator	`<`	Less than
	`>`	Greater than
	`<=`	Less than or equal to
	`>=`	Greater than or equal to
	`=`	Equal
	`==`	Equal
	`!=`	Not equal
	`<>`	Not equal
	`%%`	Regular expression match
Add	`+`	Add
	`-`	Subtract
	`|`	Bitwise OR
	`^`	Bitwise XOR
Multiply	`*`	Multiply
	`/`	Divide
	`%`	Modulus
	`&`	Bitwise AND
	`<<`	Bit shift left
	`>>`	Bit shift right, sign bit fill
	`>>>`	Bit shift right, zero fill
Unary	`!`	Logical NOT
	`-`	Negation
	`~`	Bitwise NOT

Some of the constraint operators behave differently on different data types. The operators **=**, **!=**, **<=** and **>=** have an epsilon when used on floats or doubles. The epsilon has a default value (`0.000001`) for both floats and doubles, but this default value can be changed. For example:

```
x:double y:double (x!=y);
```

Means: Read two doubles, and make sure that they're not within epsilon of each other.

Strings have a special operator, `%%`, which means "Match regular expression". None of the other comparator operators will work on a string. For example:

```
s:string (s %% "[ABC]+");
```

Means: Read a string of As, Bs and Cs.

Repeating Patterns

Of course, most data sets don't consist of just isolated data, most have patterns that repeat. In VIVA, there are two kinds of repetition: repetition across a line (or lines) of data, and repetition within a line of data. Repetition across a line (or lines) is specified within curly braces ('**{}**'), and repetition within a line is specified within angle brackets ('**<>**'). A sequence within braces ('**{}**') will repeat until end-of-file, and a sequence within brackets ('**<>**') will repeat until end-of-line. For example:

```
{
  x y (0<x, 0<y);
}
```

Means: Read pairs of positive integers, one pair per line, until EOF.

```
<x (0<x)>;
```

Means: Keep reading positive integers until EOLN.

Terminating Conditions

Each of the repeating structures can be given a terminating condition. VIVA supports two kinds of terminating conditions: by Count, and by Sentinel Value. Both are specified immediately after the opening bracket/brace, within straight brackets ('**[]**').

Terminating Condition: Count

To specify a particular number of repetitions, follow the opening bracket/brace with:

```
[* integer expression ]
```

The integer expression may be any mathematical formula built with VIVA constructs that evaluates to an integer. Any previously input variable may be used. For example:

```
n (0<n<=20);
{[*n]
  x y (0<x, 0<y);
}
```

Means: Read an integer **n**, **0**<**n**≤**20**. Then, read **n** lines of **x**'s and **y**'s.

```
<[*3] x y (-100<x<100, -100<y<100)>;
```

Means: Read exactly three ordered pairs on a single line.

```
n (0<n<=20) <[*n] x (0<x)>;
```

Means: Read an integer **`n`**, `0`<**`n`**≤`20`, and then continue to read **`n`** positive integers on the same line.

Terminating Condition: Sentinel Value

To specify a particular pattern which marks the end of the input sequence, follow the opening bracket/brace with:

```
[= pattern ]
```

The pattern may be any single-line VIVA pattern, including **<>** repetition. VIVA will read the input and test against the sentinel pattern. If the sentinel pattern matches, VIVA will exit the repetition. If the sentinel pattern fails, VIVA will reset the input to its state before trying to match the sentinel pattern, and perform another repetition. For example:

```
{[= x y (x=y=0)]
  x y (0<x, 0<y);
}
```

Means: Keep reading lines with pairs of positive integers until you read a pair of 0s.

```
<[= s:string (s %% "END")] s:string>;
```

Means: Read space-separated strings on a single line until you encounter the string "END".

Subscripts and Counts

Most data sets contain multiple values. VIVA can compare different values of the same variable with subscripts, specified by following a variable in a constraint with straight brackets ('**`[]`**'). Subscripts are zero-based, and they reset each time a repetition begins. For example:

```
x[0]
```

Means: Get the first value read for **`x`** in this scope.

The count of the number of values of that variable that have been read so far is also available, by following the variable name with a pound/hash mark ('**`#`**'). For example:

```
x#
```

Means: The number of values read for `x` so far.

```
{ x (0<=x<=5000, x#=1 || x[x#-2] < x[x#-1]); }
```

Means: Read values of **`x`** between `0` and `5000`. If there's more than one, make sure that the last one is bigger than the next-to-last one. In other words, make sure that the values of **`x`** are strictly increasing. Note that without the **`x#=1`** construct, this constraint would suffer a subscript-out-of-bounds error on the first iteration.

Cumulative Conditions

Sometimes, there is a need to check a constraint based on all of the input values taken together, not just the individual values. VIVA can do this by putting a constraint in straight brackets (‘**[]**’) at the end of either of the repeating constructs, **{}** or **<>**. For example:

```
<[*3]
    x y (-100<=x<=100, -100<=y<=100)
[(y[1]-y[0])*(x[2]-x[0]) != (y[2]-y[0])*(x[1]-x[0])]>;
```

Means: Read a line with six integers, in three (**x**,**y**) pairs, and make sure that they don’t violate the constraint.

That constraint ensures that point P0 is not equal to point P1, and that P0 is not equal to P2, and that the slope from P0 to P1 is not the same as from P0 to P2, so that the three points are not collinear. In other words, it ensures that the three points form a triangle.

Functions

VIVA has a number of useful built-in functions which can be used in constraints and expressions. There is also a capability for users to add their own functions, which will be discussed in a later section. VIVA has two types of functions: Scalar functions and Vector functions. Scalar functions take simple values as parameters. Vector functions take vectors of values as parameters, but still produce a scalar as a result. Some examples of scalar functions:

```
length(s)
```

The length of a string. Will generate an error if **s** is not a string.

```
distance(x1,y2,y1,y2)
```

Distance from (**x1**,**y1**) to (**x2**,**y2**)

```
feps(f), deps(d)
```

These set the epsilon for floats or doubles. They always return **true**, so they can be used like this:

```
x:double y:double (deps(1e-10), x!=y);
```

Setting an epsilon stays until the next change. So, in this case, double epsilon will be **1e-10** until the next **deps()** call, regardless of scope.

Vector Functions are functions that work on entire “vectors” of values in the tightest enclosing context. All other values are considered scalars. Vector functions make the most sense in Cumulative Constraints, but they can be used anywhere.

For example:

```
{ x (x>0); [sum(x)<10000] }
```

Read positive integers, one per line, until EOF. Make sure that their sum does not exceed **10000**.

```
{ x (x>0); [unique(x)] }
```

Read positive integers, one per line, until EOF. Make sure that no integer appears twice. The **unique()** function takes any number (>0) of arguments of any type. It's the combination that's checked for uniqueness.

For example:

```
{ x y (0<x, 0<y); [unique(x,y)] }
```

With this data:

```
5 6
5 5
6 5
6 6
6 5
```
🡐 **This is the line that will violate the constraint.**

Another vector function example:

```
{ x (x>0, sum(x)<5*x#); }
```

Read positive integers, one per line, until EOF. Make sure that, at every point, the sum of the **x**'s is less than **5** times the number of **x**'s read so far.

Vector Functions work over the closest enclosing scope, regardless of parameters.

For example:

```
{[*3]
    n (0<=n<=10);
    {[*5]
        m (0<=m<=10);
    [sum(m) + sum(n) + sum(6) + sum(n#) + sum(m#) < 10000]}
[sum(4) + sum(n+m) < 100]}
```

Construct	Value
sum(m)	sum of **m** values, as expected.
sum(n)	5 * the current **n** value
sum(6)	5 * 6 = 30
sum(n#)	5 * the current count of the outer scope (1, 2 or 3, so it's 5, 10 or 15)
sum(m#)	5 * 5 = 25
sum(4)	3 * 4 = 12
sum(n+m)	Parsing error, since **m** is not in the outer scope.

Fixed Width Fields

Rather than space-separated tokens, some problems specify their data in terms of fixed-width fields. VIVA has a construct to handle this: in the pattern, follow the variable name with ‘`@[` ***`integer expression`*** `]`’. If the integer expression is omitted, VIVA will read until the end-of-line. For example:

```
{ x@[5] y@[5] (0<x, 0<y); }
```

Means: Read pairs of positive integers, one pair per line, until EOF. Each integer is in a 5 character field. With the data:

```
••••1••••2
•••34••546
8273648596
```

This pattern would read **`x=1`**, **`y=5`**, then **`x=34`**, **`y=546`**, and finally **`x=82736`**, **`y=48596`**.

```
{ line:string@[]; }
```

Means: Read full lines of text until EOF.

VIVA also has a way of specifying a fixed-width field of whitespace. Instead of a variable name, insert ‘`@` ***`integer expression`*** `@`’ into the pattern. For example:

```
{ a:string@[5] @1@ y:string@[5]; }
```

Means: Read pairs strings, one pair per line, until EOF. Each string is in a 5 character field, and there is 1 character between them which is ignored. The content of the whitespace field is checked to see if it’s anything but whitespace, and an error is reported if so.

Token Image

While most constraints are concerned with the value of an input, sometimes it is desirable to check the image of an input – that is, the format of the text itself rather than the value. This token image is available by putting a dollar sign (‘**`$`**’) after the variable name in a constraint. Note that this construct is not compatible with subscripts. For example:

```
x:double (x$ %% "(([1-9]\\d*)|0)\\.\\d\\d]");
```

Means: Read a double, and make sure that it has exactly two decimal places, and that it doesn’t start with a zero unless there’s only one digit to the left of the decimal point.

```
x@[5] (rjust(x$));
```

Means: Read an integer in a 5 character field, and make sure that it’s right-justified.

Parameters

There are several parameters governing VIVA's behavior which can be set within a VIVA pattern. Parameter values may be set at the beginning of the file, before any patterns, using this syntax:

`# name1=value1 name2=value2 … #`

There can be at most one set of **##**'s in the file, and it must be the first construct in the file. For example:

`# feps=0.01F deps=1e-7 #`

Means: Set the floating point epsilon to 0.01, and the double epsilon to 0.0000001.

The following table describes the parameters available:

Name	Type	Default	Acceptable	Meaning
`feps`	float	`0.000001F`	`0.0 ≤ feps ≤ maxfloat`	Float Epsilon
`deps`	double	`0.000001`	`0.0 ≤ deps ≤ maxdouble`	Double Epsilon
`ignoreeoln`	string	`"false"`	`"true", "false", "t", "f", "yes", "no", "y", "n", "1", "0"`	Don't complain about end-of-line in a **<>** construct with a terminating condition (allow the data to run over multiple lines)
`ignoreblanks`	string	`"false"`	`"true", "false", "t", "f", "yes", "no", "y", "n", "1", "0"`	Don't complain about extra blanks
`maxerrs`	integer	`25`	`0 ≤ maxerrs ≤ maxinteger`	The maximum number of errors allowed before VIVA gives up
`eolnstyle`	string	`"system"`	`"system", "windows", "linux", "mac"`	Determine the acceptable line separator. `windows` = CRLF `linux` = LF `mac` = CR `system` = Use the line separator for the operating system where VIVA is running
`eofstyle`	string	`"system"`	`"system", "windows", "linux", "both"`	Determine whether the last line should end with a newline. `windows` = No `linux` = Yes `both` = Accept both `system` = Use the convention for the operating system where VIVA is running

Adding a Function

VIVA offers a wide range of standard functions, but should you need to add one of your own, VIVA provides a mechanism for doing this.

First, decide whether your new function will be a Scalar function or a Vector function. Remember, Scalar functions operate on individual values, whereas Vector functions operate on lists of values. You will need to implement one of two interfaces, depending on your choice: **`ScalarFunction`** or **`VectorFunction`**. There is also an abstract class, **`ArithmeticFunction`**, which is an implementation of **`ScalarFunction`**. If your function takes a single numeric parameter and returns a double, then extending **`ArithmeticFunction`** is much simpler than implementing a **`ScalarFunction`**.

Both **`ScalarFunction`** and **`VectorFunction`** extend another interface called **`Function`**. This is because they share some common methods. However, you should not implement **`Function`**. You should implement either **`ScalarFunction`** or **`VectorFunction`**, or extend **`ArithmeticFunction`**.

The following methods are common to both **`ScalarFunction`** and **`VectorFunction`**:

```
public String getName();
```

This method should return your function's name, as it would appear in a VIVA pattern.

```
public String getUsage();
```

This method should return a helpful String to explain how your function should be used. It will be given as part of the parser error if a user misuses your function.

```
public Class<?> getReturnType( Class<?> params[] );
```

This method should examine the types of the parameters that will be passed, and determine the type that your function will return. It should return one of: **`Double.class`**, **`Float.class`**, **`Integer.class`**, **`Long.class`**, **`String.class`**, **`Boolean.class`**, or **`null`** if the parameters are in error.

The difference between a **`ScalarFunction`** and a **`VectorFunction`** lies in the **`run()`** method. In a **`ScalarFunction`**, the **`run()`** method looks like this:

```
public Object run( VIVAContext context,
     List<Object> parameters ) throws Exception;
```

There will be one value passed for each of the parameters given.

In a **`VectorFunction`**, the **`run()`** method looks like this:

```
public Object run( VIVAContext context,
     List<List<Object>> parameters ) throws Exception;
```

The parameters will be a list of rows, and reach row will have one value for each parameter given. For example, with the pattern:

```
{x y; [unique(x,y)]}
```

And the data:

```
1 2
3 4
1 3
4 5
6 3
```

The **parameters** lists of the **run()** method of the **unique()** function will get:

```
[ [1, 2], [3, 4], [1, 3], [4, 5], [6, 3] ]
```

Whether your function is a **ScalarFunction** or a **VectorFunction**, your **run()** method should throw an **Exception** if the values passed are illegal (e.g. a negative value to **sqrt()**).

If your function takes a single numeric argument and returns a double, then extending **ArithmeticFunction** will be much simpler. **ArithmeticFunction** implements most of the functionality necessary for such a function. There are only two things you must do: set the **name** variable in a constructor, and implement the **implementation()** method. The **name** variable is a **String** in ArithmeticFunction, so setting it is a simple a matter of assigning it a value in your constructor:

```
public NewFunction extends ArithmeticFunction
{
    name = “newf”;
}
```

The implementation method looks like this:

```
protected abstract double implementation( double parameter )
    throws Exception;
```

Just return the computed value, or throw any necessary **Exception**.

There is a pair of very useful static methods on the **ArithmeticFunction** class:

```
public static void nanCheck( double x, String how )
    throws Exception
public static void nanCheck( float x, String how )
    throws Exception
```

The **nanCheck()** methods that check the number parameter for **NaN** (Not a Number), **Infinity**, and **–Infinity**. If the result is any of those, **nanCheck()** throws the appropriate **Exception**. The **String** parameter **how** is used in the **Exception** to tell the user exactly how the **NaN**, **Infinity** or **–Infinity** came about. These methods are public, so they can be used in any function. They are not limited to extensions of **ArithmeticFunction**.

Once your function is written, installing it is very easy. Simple call one of the **`addFunction()`** methods on a VIVA object with a new instance of your function.

```
public void addFunction( ScalarFunction function )
public void addFunction( VectorFunction function )
```

`ArithmeticFunction` is an implementation of **`ScalarFunction`**. That means that **`addFunction()`** will have no difficulties with any class you implement that extends **`ArithmeticFunction`**.

Install the new function before performing any other action, such as parsing a pattern or testing an input file. Note that in order to add a function, you must write a Java program to drive VIVA. There is currently not a way to install new functions when using command-line VIVA or the VIVA GUI.

Appendix 1: Standard Function Reference

The following functions are standard in VIVA. Note that **`numeric`** means any numeric type: **`integer`**, **`long`**, **`float`** or **`double`**.

Scalar Functions

Signature	Returns	Description
`acos(numeric)`	double	ArcCosine
`asin(numeric)`	double	ArcSine
`atan(numeric)`	double	ArcTangent
`atan2(dy,dx)` **`dy`** and **`dx`** can be any numeric type	double	ArcTangent of **`dy`**/**`dx`**, with special cases handled. Equivalent to Java's **`atan2()`**
`concat(arg1,arg2,...)` can take any number of parameters of any type	string	String concatenation of **`toString()`** of all arguments.
`cos(numeric)`	double	Cosine
`cosh(numeric)`	double	Hyperbolic Cosine
`distance(x1,y1,x2,y2)` the four parameters can be any numeric type	double	Distance from (**`x1`**,**`y1`**) to (**`x2`**,**`y2`**)
`exp(numeric)`	double	Exponential (e^x)
`if(boolean,arg1,arg2)` **`arg1`** and **`arg2`** can be of any type, but they must be of the SAME type.	type of args	If the **`boolean`** argument is true, returns **`arg1`**, otherwise returns **`arg2`**.
`length(string)`	integer	Length of the string
`ljust(string)`	boolean	Returns **`true`** if the string is left-justified (i.e. there are no leading blanks)
`ln(numeric)`	double	Natural logarithm
`log10(numeric)`	double	Logarithm base 10

`log2(numeric)`	double	Logarithm base 2
`pow(numeric, numeric)`	double	Power – first arg to the power of second
`rjust(string)`	boolean	Returns **`true`** if the string is right-justified (i.e. there are no trailing blanks)
`sin(numeric)`	double	Sine
`sinh(numeric)`	double	Hyperbolic Sine
`sqrt(numeric)`	double	Square Root
`tan(numeric)`	double	Tangent
`tanh(numeric)`	double	Hyperbolic Tangent
`test(arg1,arg2,...)` can take any number of parameters of any type	boolean	Debugging function which simply prints out its arguments to the output stream. Always returns **`true`**.
`todegrees(numeric)`	double	Convert Radians to Degrees
`todouble(arg)`	double	Convert to double
`tofloat(arg)`	float	Convert to float
`tointeger(arg)`	integer	Convert to integer
`tolong(arg)`	long	Convert to long
`toradians(numeric)`	double	Convert Degrees to Radians
`tostring(arg)`	string	Convert to string

Vector Functions

Signature	Returns	Description
`concatall(arg1,arg2,...)` can take any number of parameters of any type	string	Concatenates all arguments across all rows.
`count(boolean)`	integer	Counts all of the rows for which the boolean expression is true.
`decreasing(numeric)`	type of the argument	Returns **`true`** if the sequence of numbers is strictly decreasing. Does NOT use epsilon.
`increasing(numeric)`	type of the argument	Returns **`true`** if the sequence of numbers is strictly increasing. Does NOT use epsilon.
`nondecreasing(numeric)`	type of the argument	Returns **`true`** if the sequence of numbers is nondecreasing. Uses epsilon for floats and doubles.
`nonincreasing(numeric)`	type of the argument	Returns **`true`** if the sequence of numbers is nonincreasing. Uses epsilon for floats and doubles.
`sum(numeric)`	type of the argument	Summation.
`testall(arg1,arg2,...)` can take any number of parameters of any type	boolean	Debugging function which simply prints out its arguments for all rows to the output stream. Always returns **`true`**.
`unique(arg1,arg2,...)` can take any number of parameters of any type	boolean	Returns **`true`** if none of the rows in the data are duplicates.

Appendix 2: VIVA Pattern Examples

The following examples are taken from the 2010 Southeast USA Regional Contest of the ICPC. The input specification has been extracted from the problem statement, verbatim, and the corresponding VIVA pattern is shown.

A: Balloons

There will be several test cases in the input. Each test case will begin with a line with three integers:

N A B

Where **N** is the number of teams (1 ≤ **N** ≤ 1,000), and **A** and **B** are the number of balloons in rooms A and B, respectively (0 ≤ **A**,**B** ≤ 10,000). On each of the next N lines there will be three integers, representing information for each team:

K DA DB

Where **K** is the total number of balloons that this team will need, **DA** is the distance of this team from room A, and **DB** is this team's distance from room B (0 ≤ **DA**,**DB** ≤ 1,000). You may assume that there are enough balloons – that is, Σ(**K**'s) ≤ **A**+**B**. The input will end with a line with three 0s.

```
{[= n a b (n=a=b=0)]
    n (1<=n<=1000) a b (0<=a<=10000, 0<=b<=10000);
    {[*n]
        k da db (0<=da<=1000, 0<=db<=1000);
    [sum(k)<=a+b]}
}
```

B: Bit Counting

There will be several test cases in the input. Each test case will consist of three integers on a single line:

LO HI X

Where **LO** and **HI** (1 ≤ **LO** ≤ **HI** ≤ 10^{18}) are the lower and upper limits of a range of integers, and **X** (0 ≤ **X** ≤ 10) is the target value for K. The input will end with a line with three 0s.

```
{[= lo:long hi:long x (lo=hi=x=0)]
    lo:long hi:long (1<=lo<=hi<=1000000000000000000L)
        x (0<=x<=10);
}
```

C: Data Recovery

There will be several test cases in the input. Each test case will begin with one line containing two integers **N** and **M** (1 ≤ **N**,**M** ≤ 50), the dimension of the table. Then, the next **N** lines will each contain **M** integers. Each integer will be either between 0 and 100 inclusive, or the value -1. After the **N** lines containing table entries, there will be two more lines. The first line will contain **N** integers, each between 0 and 5,000 inclusive, indicating the sum of each row in the table, from topmost row to the bottommost row. The second line will contain **M** integers, each between 0 and 5,000 inclusive, indicating the sum of each column in the table, from the leftmost column to the rightmost column. The input will end with a line which contains two 0s.

```
{[= n m (n=m=0)]
    n m (1<=n<=50, 1<=m<=50);
    {[*n]
        <[*m] k (0<=k<=100 || k=-1)>;
    }
    <[*n] r (0<=r<=5000)>;
    <[*m] c (0<=c<=5000)>;
}
```

D: Equal Angles

There will be several test cases in the input. Each test case will consist of six integers on a single line:

AX AY BX BY CX CY

Each integer will be in the range from -100 to 100. These integers represent the three points of the triangle: (**AX**,**AY**), (**BX**,**BY**) and (**CX**,**CY**). The points are guaranteed to form a triangle: they will be distinct, and will not all lie on the same line. The input will end with a line with six 0s.

```
{[= <[*6] x (x=0)>]
    <[*3]
        x y (-100<=x<=100, -100<=y<=100)
    [(y[1]-y[0])*(x[2]-x[0]) != (y[2]-y[0])*(x[1]-x[0])]>;
}
```

E: Maximum Square

There will be several test cases in the input. Each test case will begin with two integers, **N** and **M** (1 ≤ **N**,**M** ≤ 1,000) indicating the number of rows and columns of the matrix. The next N lines will each contain M space-separated integers, guaranteed to be either 0 or 1. The input will end with a line with two 0s.

```
{[= n, m (n=m=0)]
    n m (1<=n<=1000, 1<=m<=1000);
    {[*n]
        <[*m] k (k=0 || k=1)>;
    }
}
```

F: Palindrometer

There will be several test cases in the input. Each test case will consist of an odometer reading on its own line. Each odometer reading will be from 2 to 9 digits long. The odometer in question has the number of digits given in the input - so, if the input is 00456, the odometer has 5 digits. There will be no spaces in the input, and no blank lines between input sets. The input will end with a line with a single 0.

```
{[= s:string (s %% “0”)]
    s:string (s %% “[0-9]+”, 2<=length(s)<=9);
}
```

G: Profits

There will be several test cases in the input. Each test case will begin with an integer **N** (1 ≤ **N** ≤ 250,000) on its own line, indicating the number of days. On each of the next **N** lines will be a single integer **P** (-100 ≤ **P** ≤ 100), indicating the profit for that day. The days are specified in order. The input will end with a line with a single 0.

```
{[= n (n=0)]
    n (1<=n<=250000);
    {[*n]
        p (-100<=p<=100);
    }
}
```

H: Roller Coaster

There will be several test cases in the input. Each test case will begin with a line with three integers:

N K L

Where **N** (1 ≤ **N** ≤ 1,000) is the number of sections in this particular roller coaster, **K** (1 ≤ **K** ≤ 500) is the amount that Bessie's dizziness level will go down if she keeps her eyes closed on any section of the ride, and **L** (1 ≤ **L** ≤ 300,000) is the limit of dizziness that Bessie can tolerate – if her dizziness ever becomes larger than **L**, Bessie will get sick, and that's not fun!

Each of the next N lines will describe a section of the roller coaster, and will have two integers:

F D

Where **F** (1 ≤ **F** ≤ 20) is the increase to Bessie's total fun that she'll get if she keeps her eyes open on that section, and **D** (1 ≤ **D** ≤ 500) is the increase to her dizziness level if she keeps her eyes open on that section. The sections will be listed in order. The input will end with a line with three 0s.

```
{[= N K L (N=K=L=0)]
    N (1<=N<=1000) K (1<=K<=500) L (1<=L<=300000);
    {[*N]
        F (1<=F<=20) D (1<=D<=500);
    }
}
```

I: Skyline

There will be several test cases in the input. Each test case will consist of a single line containing a single integer **N** (3 ≤ **N** ≤ 1,000), which represents the number of skyscrapers. The heights of the skyscrapers are assumed to be 1, 2, 3, …, **N**. The input will end with a line with a single 0.

```
{[= n (n=0)]
    n (3<=n<=1000);
}
```

J: Underground Cables

There will be several test cases in the input. Each test case will begin with an integer **N** (2 ≤ **N** ≤ 1,000), which is the number of points in the city. On each of the next **N** lines will be two integers, **X** and **Y** (-1,000 ≤ **X**,**Y** ≤ 1,000), which are the (**X**,**Y**) locations of the **N** points. Within a test case, all points will be distinct. The input will end with a line with a single 0.

```
{[= n (n=0)]
    n (2<=n<=1000);
    {[*n]
        x y (-1000<=x<=1000, -1000<=y<=1000);
    [unique(x,y)]}
}
```

Appendix 3: Function Code Examples

The following are examples of the Java code implementing functions. They illustrate how new functions should be written.

PowerFunction

This class implements the **`pow()`** function.

```
public class PowerFunction implements ScalarFunction
{
    public String getName()
    {
        return "pow";
    }

    public Class<?> getReturnType( Class<?>[] params )
    {
        // If there are exactly two parameters, and they’re
        // both numbers, then this function will return a
        // Double. Otherwise, return ‘null’ to indicate
        // that the function isn’t being used correctly.
        return params.length==2
            && Number.class.isAssignableFrom( params[0] )
            && Number.class.isAssignableFrom( params[1] )
                ? Double.class : null;
    }

    public String getUsage()
    {
        return "pow(number,number)";
    }

    public Object run( VIVAContext context,
        List<Object> parameters ) throws Exception
    {
        double argument =
            ((Number)parameters.get(0)).doubleValue();
        double exponent =
            ((Number)parameters.get(1)).doubleValue();

        double result = Math.pow( argument, exponent );

        ArithmeticFunction.nanCheck( result,
            "pow(" + argument + "," + exponent + ")" );

        return result;
    }
```

SumFunction

This class implements the **`sum()`** function.

```
public class SumFunction implements VectorFunction
{
    public String getName()
    {
        return "sum";
    }

    public Class<?> getReturnType( Class<?>[] params )
    {
        // If there's exactly one parameter, and it's a
        // Number, then sum() will return that type.
        // Otherwise, sum() is not being used correctly,
        // so return null.
        // If you're summing integers, it'll return an integer.
        // Summing doubles returns a double, and so on.
        return params.length==1 &&
            Number.class.isAssignableFrom( params[0] )
            ? params[0] : null;
    }

    public String getUsage()
    {
        return "sum( int or long or double or float )";
    }
    public Object run( VIVAContext context,
        List<List<Object>> parameters ) throws Exception
    {
        // We've got to be able to handle any one
        // of 4 different types
        int intsum = 0;
        long longsum = 0L;
        double doublesum = 0D;
        float floatsum = 0F;

        Class<?> type = null;
        for( List<Object> row : parameters )
        {
            // There's only one parameter per row
            Number addend = (Number)row.get( 0 );

            // Need to remember the type
            if( type==null ) type = addend.getClass();

            if( type==Integer.class )
                intsum += addend.intValue();
            else if( type==Long.class )
                longsum += addend.longValue();
            else if( type==Double.class )
                doublesum += addend.doubleValue();
            else if( type==Float.class )
                floatsum += addend.floatValue();
        }

        ArithmeticFunction.nanCheck( doublesum, "sum()" );
        ArithmeticFunction.nanCheck( floatsum, "sum()" );

        Object value = null;
        if( type==Integer.class )
            value = new Integer(intsum);
        else if( type==Long.class )
            value = new Long(longsum);
        else if( type==Double.class )
            value = new Double(doublesum);
        else if( type==Float.class )
            value = new Float(floatsum);

        return value;
    }

}
```

SquareRootFunction

This class implements the **`sqrt()`** function. It provides an example of extending the **`ArithmeticFunction`** class.

```
public class SquareRootFunction extends ArithmeticFunction
{
    public SquareRootFunction()
    {
        name = "sqrt";
    }

    protected double implementation( double parameter )
        throws Exception
    {
        if( parameter<0.0 )
        {
            throw new Exception(
                "Parameter ("+parameter+") to sqrt() is <0" );
        }

        return Math.sqrt( parameter );
    }

}
```