RPGsh User Manual

TheMohawkNinja

February 10, 2025

Ver. 0.14.0

DISCLAIMER: This project is entirely developed independantly. I am not associated in anyway with Wizards of the Coast, Paizo, or any other game development company.

Contents

1 Introduction and Basic Usage

The Roleplaying Games Shell, `rpgsh`, is an interactive and extensible shell purpose-built for augmenting player and DM gameplay for table-top RPGs like Dungeons & Dragons®, Pathfinder®, and more!

`rpgsh` provides users with capabilities similar to those found in conventional shells (e.g. `bash` or `PowerShell`) like command execution and variable assignment/modification, while also adding functionality more unique to shell environments like varying data types and variable scopes.

1.1 The Prompt

When interacting with the shell directly, you will be presented with a prompt that will look similar to the following:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$
```

The prompt contains the currently loaded character's name (*<NO_NAME>*) along with their **current/max** *(temp)* hitpoints.

As with any command line interface, you interact with the prompt by entering in either a variable or a program, followed by any operators or parameters in a space-delimited format. For example, if you want to roll a 20-sided die, you would enter the following:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ roll d20
```

The maximum size of the input buffer for the prompt is 65,535 characters. Exceeding this will throw an error.

1.2 Escape Characters

Due to the use of certain special characters in `rpgsh`, users wishing to have these characters interpreted literally will need to escape them. This can be done by prefixing them with the backslash '\' character. Additionally, special control characters like backspace '\b' are also supported via escaping.

1.3 Controls

<Printable Characters>	Print character to input buffer
Left Arrow	Move cursor one column to the left
Right Arrow	Move cursor one column to the right
Up Arrow	Cycle backwards through `rpgsh` history one step
Down Arrow	Cycle forwards through `rpgsh` history one step
Page Up	Go to first line in `rpgsh` history
Page Down	Go to last line in `rpgsh` history
Tab	Cycle forward through tab-completion matches one step
Shift+Tab	Cycle backward through tab-completion matches one step
Home	Go to beginning of input buffer
End	Go to end of input buffer
Insert	Toggle insert mode
Enter	Execute input buffer

2 Text Formatting

`rpgsh` supports common escape characters along with special text formatting strings for specifying foreground colors, background colors, font styles, and other effects to text when using programs like `print` to render the text.

2.1 Escape Characters

\a	Bell
\b	Backspace
\f	Form Feed
\n	Newline
\t	Horizontal Tab
\v	Vertical Tab

Any other character prefixed with a backslash with be interpreted literally.

2.2 Format Strings

All format strings start and end with a %. These are omitted in the table below for brevity. Note that not all terminals support all of the effects stated below.

\	Reset all effects
b	Bold
\b	No bold
i	Italic
\i	No italic
d	Dim
\d	Normal intensity (resets Bold and Italic)
u	Underline
\u	No underline
blink	Blink
\blink	No blink
r	Reverse (flips background and foreground colors)
\r	No reverse
*basic_color**	Sets text to the specified *basic_color*
*basic_color**	Unsets *basic_color*
bg*basic_color**	Sets background to the specified *basic_color*
\bg*basic_color**	Unsets bg*basic_color*
fg=*HTML_color***	Sets text to the specified *HTML_color*
\fg=*HTML_color***	Unsets fg=*HTML_color*
fg=*r,g,b*	Sets text to the specified color using 8-bit color channels
\fg=*r,g,b*	Unsets fg=*r,g,b*
bg=*HTML_color***	Sets background to the specified *HTML_color*
\bg=*HTML_color***	Unsets bg=*HTML_color*
bg=*r,g,b*	Sets background to the specified color using 8-bit color channels
\bg=*r,g,b*	Unsets bg=*r,g,b*

* Values are: black, red, green, yellow, blue, magenta, cyan, lightgray, darkgray, lightred, lightgreen, lightyellow, lightblue, lightmagenta, lightcyan, or white. Although 24-bit color support is common among modern terminals, these colors (especially the non-light and non-dark variants) should be supported.

** `rpgsh` supports all 140 of the W3C named colors for HTML, case-insensitive.

3 Program Listing

As of version 0.14.0, the following programs are available to the user when interacting with the `rpgsh` prompt:

`banner`

Displays the ASCII art logo for `rpgsh` along with a one-line description of the program and the author's signature.

`clear`

Clears the screen.

`del`

Displays the ASCII art logo for `rpgsh` along with a one-line description of the program and the author's signature.

`eval`

Prints values and evaluates operations. Implicitly called when the user enters a variable as the first parameter in the prompt.

`help`

Lists all applications available to `rpgsh` along with a brief description.

`history`

Prints `rpgsh` history.

`list`

Lists all the variables in one or all scopes.

`print`

Pretty prints variables, variable sets, and scopes.

`roll`

Dice-rolling program which supports custom lists and result counting.

`setname`

Sets which variable is used for displaying the character's name.

`version`

Prints `rpgsh` version information.

4 Variables

`rpgsh` allows the user to set, get, and modify variables. Variables are arranged in a nested hierarchy through three different scopes, and through an arbitrary number of levels within each scope. The components of an operation must be space delimited as shown below:

`$v/four = 2 + 2` **Correct**

`$v/four=2+2` **Incorrect**

Variables in `rpgsh` follow the below syntax:

$$scope[xref]type/key$$

Below describes each part in detail:

4.1 scope

A single character representing which level of the overall hierarchy is being referenced. There are three total scopes for use in `rpgsh`:

`@` Character variables. This scope encompasses all variables specific to a given character. By default, this references the currently loaded character.

These are stored in `~/.rpgsh/campaigns/<campaign>/characters/<charactername>.char`

`#` Campaign variables. This scope encompasses all variables in a given campaign. By default, this references the current campaign.

These are stored in `~/.rpgsh/campaigns/<campaign>/.variables`

`$` Shell variables. This scope encompasses all campaigns and is the broadest scope in `rpgsh`.

These are stored in `~/.rpgsh/.variables`

To assist in remembering which character represents which scope, there are two rules of thumb:

1. At least with conventional U.S. keyboard layouts, the breadth of the scope increases as you go right on the number row. Shift+2 is `@`, shift+3 is `#`, and shift+4 is `$`.
2. The `@` symbol can be thought of in the context of modern social media and text chat applications whereby it points *at* a specific person in the same way that in `rpgsh` it points *at* a specific character. The `#` symbol has the appearance of a grid, and thus can be thought of as encompassing everything on the grid of the game board. Lastly. `$` should be familiar to anyone who has used *nix scripting languages, as they also represent shell variables respective to their environment.

4.2 xref *(optional)*

A case-insensitive external reference to allow the user to get and set variables from outside the current character or campaign.

If the scope is a character variable, then the xref is the name of another character from within the current or other campaign. If the scope is a campaign variable, then the xref is the name of another campaign. This option is not available to the shell scope, as there is only one shell scope.

Note that the square brackets only need to be printed if you are using an xref.

For example, the following demonstrates accessing the `HP/Current` var-type variable from the character "kobold" from the current camapign:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @[kobold]v/HP/Current
```

Likewise, the following demonstrates accessing the `QuestsCompleted` var-type variable from the campaign "MyCampaign":

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ #[MyCampaign]v/QuestsCompleted
```

Furthermore, characters from other campaigns can be referenced by formatting the xref as *campaign/character*. For example, if I wanted to access the var-type variable "Initiative" from the character "goblin" from the campaign "MyCampaign", I would enter the following:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @[MyCampaign/goblin]v/Initiative
```

4.3 type *(optional)*

A single character representing the data type of the variable. As of version 0.14.0, the following data types have been implemented: `v` (Var), `d` (Dice), `c` (Currency), `s` (Currency System), and `w` (Wallet). Note that in all operations, the data type of the returning value will always be the same as the left-hand side (LHS) of the operation. Omission of a type will have an effect dependant on the format of the key.

For each data type described in this section, the following subsections describe various attributes associated with the given data type:

Constructors:

These describe the ways in which each data type can be created while using `rpgsh`. These necessarily include a *explicit* constructor, which is in the format of $c\{Properties\}$, where c is some lower-case character that defines which data type is being constructed, and *Properties* which are one or more numbers and/or strings of characters that make up the constructed object. Additionally, constructors may include one or more *implicit* constructors, which do not have a universal format, but make for a more human-readable means of interacting with data types.

Properties:

These describe each property as declared in the explicit constructor definition, along with noting whether or not it is optional. If more than one properties are available to be defined, they must be in a comma-delimited list, and all commas must be entered even if a given optional property is omitted.

When calling a variable, the properties of a variable can be accessed by appending `.Property` to the end of the variable.

Examples:

Examples of possible ways to construct the given data type.

Operations Table(s):

These describe what happens when you operate on a variable for the section's data type. Each cell in a table describes the result of an operation in which the left-hand side (LHS) is of the type currenly being described by that subsection of the document, the operator is the row header, and the right-hand side (RHS) is the column header.

In the case of arithmetic, assignment, and unary operators, properties affected by the operation are printed in *italics*. Cells marked with an **ERR** result in an error being thrown with no change being made to the LHS.

In the case of relational and boolean operators, the cells represent the conditions required for the operation to return true. Cells marked with an **F** will always return false. Additionally for boolean operators, the right column describes the condition(s) in which the subsection's data type will evaluate to *true*.

Operations follow PEMDAS, with operator precedence defined as:

1. `(` `)`
2. `++` `--`
3. `*` `/` `^` `%`
4. `+` `-`
5. `<` `<=` `>` `>=`
6. `==` `!=`
7. `&&`
8. `||`
9. `=` `*=` `/=` `^=` `%=` `+=` `-=`

4.3.1 Var

These are generic, lazily-evaluated variables that may contain either a string or an integer, similar to how variables in many scripting languages operate. Operations performed on var-type variables are thus dependant on whether or not the current value stored is evaluated to be a string or an integer.

Constructors:

- An integer
- A string of characters
- A string of characters wrapped in quotation marks
- v{*Value*}

Properties:

Value:
The value of the var-type variable. This can be any number or string of text. When used in an explicit constructor, quotation are not necessary for strings containing spaces.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = 3
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = three
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = "The number three"
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @v/MyVar = v{The number three}
```

Operations Tables:

LHS evaluates to Integer					
Op (Arith.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`+`	*Value* Addition	**ERR**	**ERR**	**ERR**	**ERR**
`-`	*Value* Subtraction				
`*`	*Value* Multiplication				
`/`	*Value* Division				
`^`	*Value* Exponentiation				
`%`	*Value* Modulo				
Op (Assign.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`=`	*Value* Assignment	*Value* Assign.*	**ERR**	**ERR**	**ERR**
`+=`	*Value* Addition Assign.	**ERR**			
`-=`	*Value* Subtraction Assign.				
`*=`	*Value* Multi. Assign.				
`/=`	*Value* Division Assign.				
`^=`	*Value* Exponent. Assign.				
`%=`	*Value* Modulo Assign.				
Op (Relat.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`==`	*Value* == RHS *Value*	**F**	**F**	**F**	**F**
`<`	*Value* < RHS *Value*				
`>`	*Value* > RHS *Value*				
`<=`	`<` or `==`				
`>=`	`>` or `==`				
`!=`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`

Op (Bool)	—
`&&` OR `||`	*Value* != 0
Op (Unary)	—
`++`	*Value* Increment
`--`	*Value* Decrement

LHS evaluates to String					
Op (Arith.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`+`	**ERR**	*Value* Concat.	**ERR**	**ERR**	**ERR**
`-`		**ERR**			
`*`	*Value* Multiplication				
`/`	**ERR**				
`^`					
`%`					
Op (Assign.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`=`	*Value* Assignment*	*Value* Assign.	**ERR**	**ERR**	**ERR**
`+=`	**ERR**	*Value* Concat. Assign.			
`-=`		**ERR**			
`*=`		*Value* Multi. Assign.			
`/=`		**ERR**			
`^=`					
`%=`					
Op (Relat.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`==`	**F**	*Value* == RHS *Value*	`d` `==` **	**F**	**F**
`<`		*Value* lower in alphabet to RHS *Value*	`d` `<` **		
`>`		*Value* higher in alphabet to RHS *Value*	`d` `>` **		
`<=`		`<` or `==`	`d` `<=` **		
`>=`		`>` or `==`	`d` `>=` **		
`!=`	Neg. of `==`	Neg. of `==`	`d` `!=` **	Neg. of `==`	Neg. of `==`

Op (Bool)	—
`&&` OR `||`	*Value* != "" (empty) AND *Value* != "False" (case-insensitive)
Op (Unary)	—
`++`	**ERR**
`--`	

*A warning will be thrown to indicate that the evaluated data type has changed.

**If the string is a properly formed dice implicit constructor, please see the appropriate cell for the referenced operation and RHS data type.

4.3.2 Dice

These are variables which not only can be constructed and printed in the standard RPG dice format, but operations performed on dice are meant to allow users to more intuitively interact with the dice they may need to roll throughout gameplay.

Constructors:

- *Quantity*d*Faces*[+|−]*Modifier*
- d{*Quantity*,*Faces*,*Modifier*}

Properties:

Quantity (optional):
The number of dice. If omitted, assumes a value of 1.

Faces:
The number of faces of the di(c)e.

Modifier (optional):
A modifier value which affects the total roll value. If omitted, assumes a value of 0.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = d20
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = 2d8
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = 3d6+1
```

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ @d/MyDice = d{1,20,-5}
```

Operations Tables:

Op (Arith.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`+`	*Modifier* Add.	*Quantity* Add. OR *List* Concat.***	*Quantity* Add.**	**ERR**	**ERR**
`-`	*Modifier* Sub.	*Quantity* Sub.*	*Quantity* Sub.**		
`*`	*Quantity* Multi.	**ERR**	**ERR**		
`/`	*Quantity* Div.				
`^`	**ERR**				
`%`	*Quantity* Modulo				

Op (Assign.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`=`		Assignment*	Assignment	**ERR**	**ERR**
`+=`	*Modifier* Add. Assign.	*Quantity* Add. Assign. OR *List* Concat. Assign.***	*Quantity* Add. Assign.**		
`-=`	*Modifier* Sub. Assign.	*Quantity* Sub. Assign.*	*Quantity* Sub. Assign.**		
`*=`	*Quantity* Multi. Assign.	**ERR**	**ERR**		
`/=`	*Quantity* Div. Assign.				
`^=`	**ERR**				
`%=`	*Quantity* Modulo Assign.				

Op (Relat.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`==`	**F**	`==` `d` †	All properties == All RHS properties	**F**	**F**
`<`		`<` `d` †	*Quantity* < RHS *Quantity*		
`>`		`>` `d` †	*Quantity* > RHS *Quantity*		
`<=`		`<=` `d` †	`<` OR `==`		
`>=`		`>=` `d` †	`>` OR `==`		
`!=`	Neg. of `==`	`!=` `d` †	Neg. of `==`	Neg. of `==`	Neg. of `==`

Op (Bool)	—
`&&` OR `||`	*Quantity* > 0

Op (Unary)	—
`++`	*Modifier* Increment
`--`	*Modifier* Decrement

*If and only if the string is formatted appropriately, otherwise an error will be thrown.

**If and only if both dice have equal faces. Additionally, in the event that both dice have different modifiers, a warning will be thrown indicating that only the LHS modifier will be preserved.

***If the string is a properly formed dice implicit constructor, *Quantity* will be affected, otherwise *List* will be affected.

†If the string is a properly formed dice implicit constructor, please see the appropriate cell for the referenced operation and RHS data type.

4.3.3 Currency

These are variables which are used to handle monetary values. If the currency is part of a currency system, `rpgsh` can automatically calculate change and merge smaller denominations into larger denominations as needed.

Due to limitations in the way information about inter-currency relationships are inferrede, users should limit themselves to only referencing currency-type variables contained within the three currently loaded scopes, making sure to avoid the use of an *xref* to prevent ambiguities during runtime.

Constructors:

- c{*CurrencySystem*,*Name*,*SmallerAmount*,*Smaller*,*Larger*}

Properties:

CurrencySystem (optional):
The name of the currency system that the currency is a part of. If the game only has one currency system, this property may be omitted.

Name:
The name of the currency. It must be unique within the scope that the currency is being declared within. In most cases, this should be the Campaign scope, as currencies usually cover more than one character in a given game, whereas the Shell scope would cover all games, which would be non-ideal unless you know for a fact you will only ever play the same game.

SmallerAmount (optional):
The amount of the larger denomination needed to equal this denomination. If the game only has one currency, this property may be omitted.

Smaller (optional):
The name of the smaller denomination. If the game only has one currency, this property may be omitted.

Larger (optional):
The name of the larger denomination. If the game only has one currency, this property may be omitted.

Examples:

```
┌─[<NO_NAME>]─(0/0 (0))
└─$ #c/gold = c{dnd5e,Gold,10,Silver,Platinum}
```

Operations Tables:

Op (Arith.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`+`	*SmallerAmount* Add.	*Name* Concat.	**ERR**	`w{LHS,1,RHS,1}`	**ERR**
`-`	*SmallerAmount* Sub.	**ERR**		**ERR**	
`*`	`w{LHS,RHS}`				
`/`	**ERR**				
`^`					
`%`					

Op (Assign.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`=`	**ERR**	**ERR**	**ERR**	Assignment	**ERR**
`+=`	*SmallerAmount* Add. Assign.	*Name* Concat. Assign.		**ERR**	
`-=`	*SmallerAmount* Sub. Assign.	**ERR**			
`*=`	**ERR**				
`/=`					
`^=`					
`%=`					

Op (Relat.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`==`	**F**	**F**	**F**	All properties == all RHS properties	**F**
`<`				*Larger* == RHS *Name**	
`>`				*Smaller* == RHS *Name**	
`<=`				`<` OR `==`	
`>=`				`>` OR `==`	
`!=`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`

Op (Bool)	—
`&&` OR `||`	*Smaller* != "" (empty)

Op (Unary)	—
`++`	*SmallerAmount* Increment
`--`	*SmallerAmount* Decrement

*Or there exists a smaller or larger currency in which its' *Larger* or *Larger* respectively == RHS *Name*.

4.3.4 Wallet

A Wallet-type variable. Like a real, physical wallet, instances of this data type contain quantities of one or more Currency-type variables. These are both meant to be used as the wallet or coin purse of given character, but also as a formalized way of defining the cost of items.

Constructor:

- w{$Currency_1$,$Quantity_1$,$Currency_2$,$Quantity_2$, ... ,$Currency_n$,$Quantity_n$}

Properties:

$Currency_x$
A Currency-type variable. Unlike other properties, you do not reference this property by entering the word "Currency", but rather entering in the name of the desired currency.

$Quantity_x$
The amount of $CurrencyName_x$ in the Wallet-type variable. Note that unlike all other properties, this one cannot be accessed directly, but is printed when accessing the *Currency* property.

Examples:

```
[<NO_NAME>]-(0/0 (0))
$ @w/MyWallet = w{#c/Gold,10}
```

```
[<NO_NAME>]-(0/0 (0))
$ @w/MyWallet = w{#c/Gold,10,#c/Silver,5}
```

```
[<NO_NAME>]-(0/0 (0))
$ @w/MyWallet = w{#c/Gold,10,#c/Silver,5,#c/Copper,3}
```

Operations Tables:

Due to the fact that, unlike other data types, Wallet-type variables contain an arbitrary number of its' two properties, references to a given property (unless stated otherwise) in the below table refer to each instance of said property in a given wallet. This can be thought of as "for each *Property* in the wallet, do *something*."

Op (Arith.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`+`	**ERR**	**ERR**	**ERR**	*Currency* Add.	**ERR**
`-`				*Currency* Sub.	
`*`	*Currency* Multi.			**ERR**	
`/`	*Currency* Div.*				
`^`	**ERR**				
`%`	*Currency* Mod.				

Op (Assign.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`=`	**ERR**	**ERR**	**ERR**	`=` `w{RHS,1}`	**ERR**
`+=`				*Currency* Add. Assign.	
`-=`				*Currency* Sub. Assign.	
`*=`	*Currency* Multi. Assign.			**ERR**	
`/=`	*Currency* Div. Assign.*				
`^=`	**ERR**				
`%=`	*Currency* Mod. Assign.				

Op (Relat.)	`v` *(Integer)*	`v` *(String)*	`d`	`c`	`w`
`==`	**F**	**F**	**F**	`==` `w{RHS,1}`	**F**
`<`				`<` `w{RHS,1}`	
`>`				`>` `w{RHS,1}`	
`<=`				`<` OR `==`	
`>=`				`>` OR `==`	
`!=`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`	Neg. of `==`

Op (Bool)	—
`&&` OR `||`	Quantity of at least one *Currency* > 0

Op (Unary)	—
`++`	Smallest *Currency* of each CurrencySystem Inc.
`--`	Smallest *Currency* of each CurrencySystem Dec.

*Just as in real banking, the division of quantities of currencies may be lossy, as currencies are not infinitely divisible. Thus, when dividing the quantities in a wallet, any remainder after making change down to the lowest denomination is lost. For example, suppose we have a currency system containing Dollars and Pennies, where 1 Dollar equals 100 Pennies, and we have a wallet containing 1 Dollar. If I divide that wallet by 3, I would end up with 33 Pennies remaining in my wallet, with the remaining 1/3 of a Penny being lost.

4.4 key *(optional)*

The key is a case-insensitive, forward slash-delimited string representing the variable or variables within a given scope that the user intends to reference. In a manner analagous to the folder structure in Linux or Windows, the forward slash delimiting allows a hierarchical organization of data within `rpgsh`. When referencing individual keys, the type specifier tells `rpgsh` to specifically use the variable associated with the specified key and data type. If the type specifier is omitted, `rpgsh` will check each data type in the following order, using the first match: `v`, `d`, `w`, `c`, `s`. In most cases, the type can be omitted when printing variables, as it would be unlikely for two variables of differing data types to have the same key.

For example, if you want to print the variable `Strength/Modifier` var-type variable from your current character, both of the below syntaxes are correct:

```
[<NO_NAME>]-(0/0 (0))
$ @v/Strength/Modifier
```

```
[<NO_NAME>]-(0/0 (0))
$ @/Strength/Modifier
```

If the key ends in a forward slash, `rpgsh` will print the entire variable set, starting with the root key and including all downstream keys. For example, if you wanted to print both `@v/Strength` and `@v/Strength/Modifier`, you would simply need to enter:

```
[<NO_NAME>]-(0/0 (0))
$ @/Strength/
```

When printing entire sets of variables, both the key and value for each variable in the set are printed to the screen, with `::` delimiting between both the key and value, and each key/value pair. This format is mainly for use as an easily-parsable string in the event a variable set is passed as an argument to a program. For a more human-readable format, it is recommended to use the `print` program to pretty print such data.

Additionally, when printing sets of data, the type specifier acts as a filter allowing a user to only print the variables of the specified data type. For example, if the user wanted to print just the dice-type variables on their currently loaded character, they simply enter:

```
[<NO_NAME>]-(0/0 (0))
$ @d/
```

Lastly, like with singlular variables, sets can also have operations performed on them. However, set operations can only be performed with other sets, and only `+`, `+=`, `-`, `-=` are supported. The following table details set operations:

Op (Set)	—
`+`	Adds the RHS root key and all downstream keys to the LHS, printing the result.
`+=`	Adds the RHS root key and all downstream keys to the LHS, saving the result.
`-`	Removes the RHS root key and all downstream keys from the LHS, printing the result.
`-=`	Removes the RHS root key and all downstream keys from the LHS, saving the result.