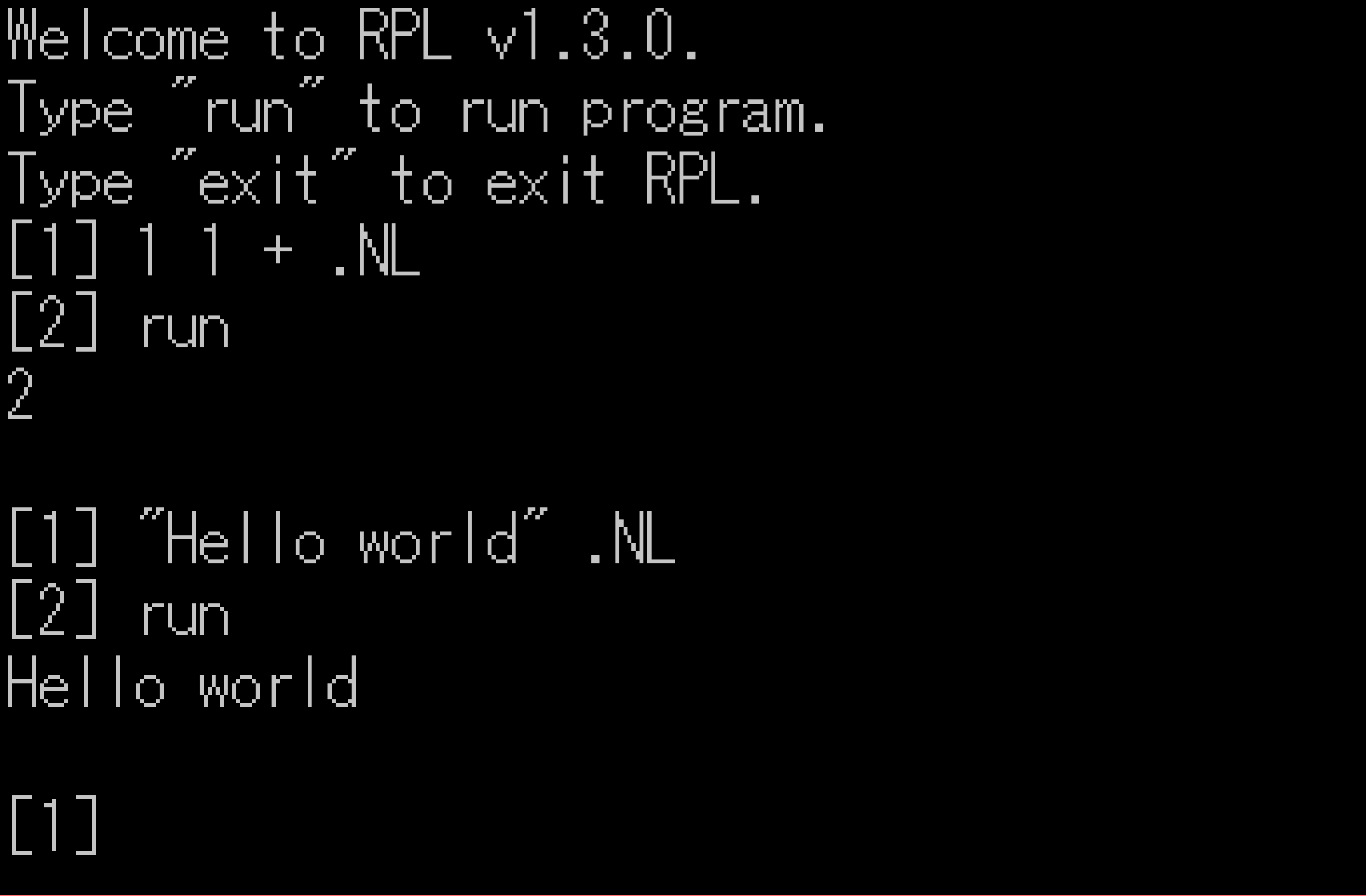
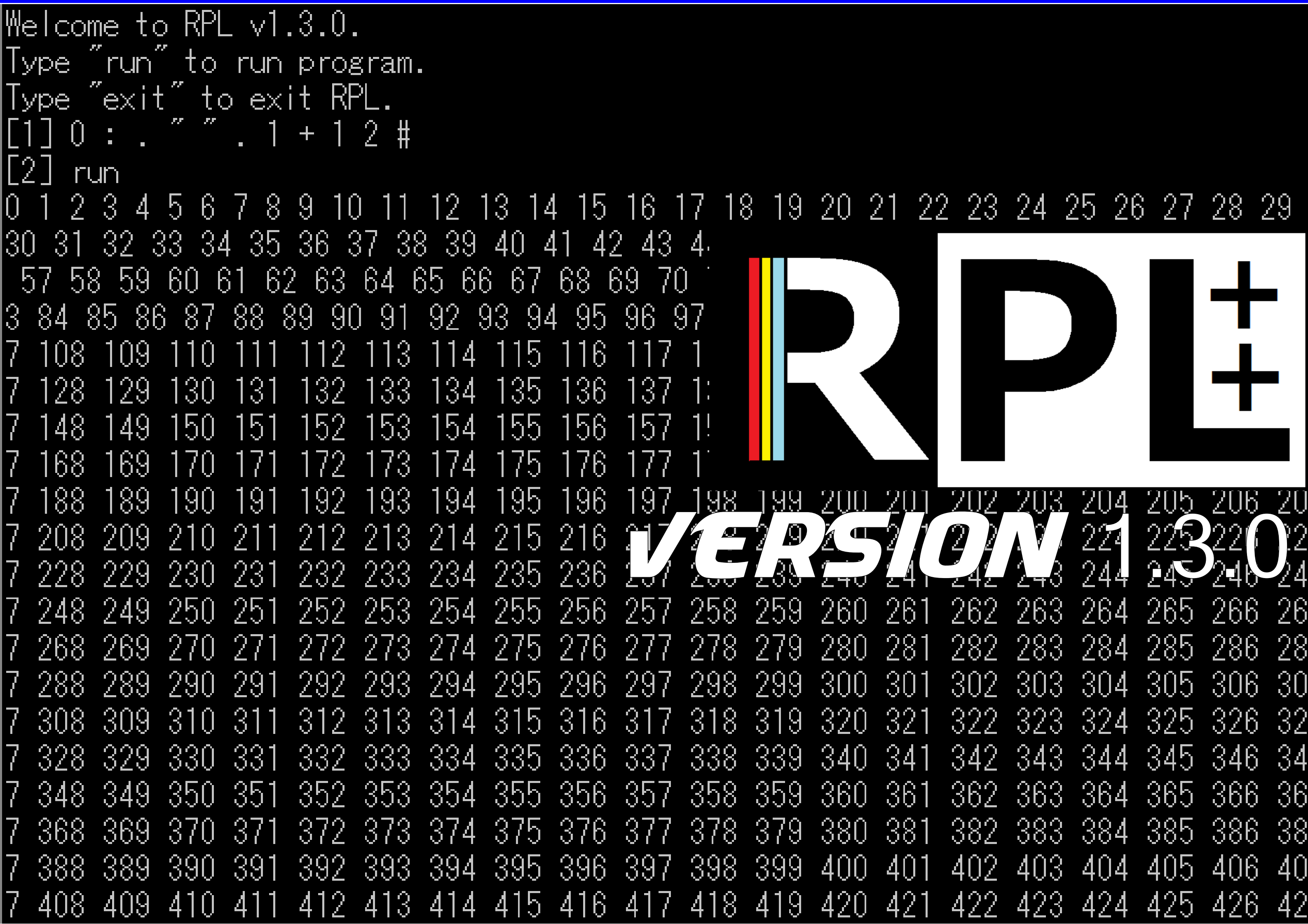


```
Welcome to RPL v1.3.0.
Type "run" to run program.
Type "exit" to exit RPL.
[1] 1 1 + .NL
[2] run
2

[1] "Hello world" .NL
[2] run
Hello world

[1]
```

Reference Handbook



```
Welcome to RPL v1.3.0.
Type "run" to run program.
Type "exit" to exit RPL.
[1] 0 : . " " . 1 + 1 2 #
[2] run
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42 43 4
 57 58 59 60 61 62 63 64 65 66 67 68 69 70
3 84 85 86 87 88 89 90 91 92 93 94 95 96 97
7 108 109 110 111 112 113 114 115 116 117 1
7 128 129 130 131 132 133 134 135 136 137 1
7 148 149 150 151 152 153 154 155 156 157 1
7 168 169 170 171 172 173 174 175 176 177 1
7 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 20
7 208 209 210 211 212 213 214 215 216
7 228 229 230 231 232 233 234 235 236
7 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 26
7 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 28
7 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 30
7 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 32
7 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 34
7 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 36
7 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 38
7 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 40
7 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 42
```

RPL++

VERSION 1.3.0

RPL++

Reference Handbook

TABLE OF CONTENTS

CHAPTER 1. THE RPL++ LANGUAGE: *Syntax*

RPL++ uses Reverse Polish Notation.
So, 1 + 1 in RPL++ is:

```
1 1 +
```

And, RPL++ is stack-based.
For example, stack when you run `1 2 +` is like:

Word: 1
Stack:

1

Word: 2
Stack:

2
1

Word: +

RPL++ pops top 2 values from the stack
(Because + operator takes 2

values), so
in this case, RPL++ pops 2, and 1, adds them, and pushes the result (3) to the stack.

Stack:

3

1 – 1. Types

RPL++ has only 5 types, String, Number, undef, notnum and Array.

Name	String	Number	undef	notnum	Array
Example	"String"	123	undef	notnum	See Chapter 5

1 – 2. Comment

There are 2 comment types, single-line comment, and multi-line comment.

Name	Single-line	Multi-line
Example	// Comment	/* Comment */

CHAPTER 2. THE RPL++ LANGUAGE: *Mathematical Operators*

The operation of each operator is described with a single notation. The following symbold are used:

( )	=	contents of
↑	=	"is popped from stack"
↓	=	"is pushed from stack"
2↑	=	"is popped from 2nd stack"
2↓	=	"is pushed from 2nd stack"
^	=	the value of the top of the stack
A	=	boolean AND
V	=	boolean OR
~~V~~	=	exclusive OR
~	=	boolean not
R[ ],op	=	Reverses the values, and does op

Symbol	
+	Addition

`-`	Subtraction
`*`	Multiplication
`/`	Division
`%`	Remainder
`_`	Misc
`?`	Random
`!`	Factorial

+

Operation: ↓(↑)+(↑)

−

Operation: ↓R[(↑),(↑)],−

*

Operation: ↓(↑)×(↑)

/

Operation: ↓R[(↑),(↑)],/

%

Operation: ↓R[(↑),(↑)],mod

—

Operation: If (↑) is 0

```
    ↓floor((↑))
```

If (↑) is 1

```
    ↓ceil((↑))
```

If (↑) is 2

```
    ↓round((↑))
```

If (↑) is 3

```
    ↓R[(↑),(↑)],pow
```

?

Description: It generates a random number $0 \leqq n < 1$

!

Operation: `factorial((↑))`

CHAPTER 3. THE RPL++ LANGUAGE: *Input-Output Operators*

Symbol	
`.`	Console output
`.NL`	Console output with newline
`.?`	Gets the input from stdin
`NL`	Outputs newline
`#>`	File Input
`#<`	File Output
`#?`	Checks if file exists

.

Operation: `print((↑))`

.NL

Operation: `println((↑))`

.?

Operation: `↓readline()`

NL

Operation: `println()`

#>

Operation: `↓open((↑)).read()`

#<

Operation:

```
open((↑)
      .write((↑))
```

#?

Operation:

```
↓file_exists((↑))
```

CHAPTER 4. THE RPL++ LANGUAGE: *Bitwise & Logical Operators*

Add period at the end of the logical operator to make it bitwise operator.

Symbol	
&	Logical AND
|	Logical OR
^	Logical XOR
~	Logical NOT

&

Operation: ↓ (↑) & (↑)

|

Operation: ↓ (↑) | (↑)

^

Operation: ↓ (↑) ^ (↑)

~

Operation: ↓ ~ (↑)

CHAPTER 5. THE RPL++ LANGUAGE: *Array Operators*

Symbol	
`]`	Creates an array
`[`	Extracts the array
`] [`	Gets the length of the array
`@`	Gets the value of the array
`[$]<`	Changes the value of the array
`[$]^`	Removes the top value of the array
`[$]-`	Creates a zero-filled array
`[$]+`	Concatenates the 2 arrays

]

Description:	Pushes an array with length that is specified by the 1st popped value to the stack.
Example:	`1 2 3 3 ]`
Result:	Array [1, 2, 3] will be pushed to the stack.

[

Description:	Extracts the array to the stack.
Example:	`1 2 3 3 ] [`
Result:	1 (it'll be pushed first), 2, and 3 will be pushed to the stack.

```
] [
```

Description: Pushes the length of the array to the stack.

```
@
```

Description: Gets the value of the index that is specified by the 1st popped value of the array that is specified by the 2nd popped value

```
[$]<
```

Description: Changes the index that is specified by the 2nd popped value of the array that is specified by the 3rd popped value to the 1st popped value, and pushes the changed array to the stack.

[$]^

Description: Removes the value of the array that is specified by the 1st popped value.

[$]-

Description: Creates an array with the length that is specified by the 1st popped value.

[$]+

Description: Concanecates the 2 arrays that is specified by the 2nd popped value, and the 1st popped value.

CHAPTER 6. THE RPL++ LANGUAGE: *Stack Operators*

Symbol	
`:`	Duplicates the value
`\`	Removes the value
`2>`	Gets the value from the 2nd stack
`2<`	Pushes the value to the 2nd stack
`<>`	Reverses the multiple values

:

Operation: ↓(^)

\

Operation: ↑

2>

Operation: 2↓(↑)

2<

Operation: ↓(2↑)

<>

Operation: **Reverses the values that is length specified with the 1st popped value.**

CHAPTER 7. THE RPL++ LANGUAGE: *Comparison Operators*

In RPL++, comparison operators are used like other operators too.

Symbol	
`[EQ]`	**E**qual to
`[NE]`	**N**ot equal to
`[LT]`	**L**ess **t**han
`[LE]`	**L**ess than or **e**qual to
`[GT]`	**G**reater **t**han
`[GE]`	**G**reater than or **e**qual to

CHAPTER 8. THE RPL++ LANGUAGE: *Misc Operators*

Symbol	
=	Defines the variable
$	Converts the string to a number
$>?	Converts the number to a character (U+xxxx)
:<	Goes back to the recursive call

=

Description:	Defines the variable that is specified by the 1st popped value if it's not defined, and assigns the value that is specified by the 2nd popped value to the variable.
Example:	`3.14 "pi" =`
Result:	Variable pi will be defined, and assigned 3.14 to it.

$

Description:	Converts the 1st popped value to the number.

`$>?`

Description:	Converts the 1st popped value to a character.
Example:	`0x21 $>?`
Result:	U+0021 is Exclamation Mark, so `!` will be pushed.

`:<`

Description:	Goes back to the latest recursive call (See Chapter 9).

CHAPTER 9. THE RPL++ LANGUAGE: *Statements*

Symbol	
`.IM`	Imports the library
`.IMS`	Imports the standard library
`#`	Goto
`;`	Goto with the condition
`.FN`	Defines a function
`.CALL`	Calls the function
`_CALL`	Same with `.CALL`
`;label`	Defines an one-line skip label
`:label`	Defines a label
`>label`	Jumps to the label
`?>label`	Jumps to the label with the condition
`:>label`	Jumps to the label with the recursive mode
`?:>label`	Jumps to the label with the recursive mode

`_name_`	Defines a “skipper”
`_>name`	Jumps to the skipper
`_?>name`	Jumps to the skipper with the condition
`wd-*`	Worddef
`?( ).`	If statement
`?!( ).`	If not statement

.IM

Description: Imports the library.

.IMS

Description: Imports the standard library.

#

Description: Jumps to the line that is specified by the 2nd popped value, and word that is specified by the 1st popped value.

;

Description: If the 3rd popped value is 1, it does the almost same thing with #

.FN

[DEPRECATED]

Description:	Defines a function. It's deprecated, so no more description.

.CALL

[DEPRECATED]

Description:	Calls the function. It's deprecated, so no more description.

_CALL [DEPRECATED]

Description:	Calls the function. It's deprecated, so no more description.

;label

Description:	Defines an one-line skip label. Line where the one-line skip label is will be skipped when it's defined.

:label

Description:	Defines a label.

>label

Description:	Jumps to the label.

`?>label`

Description:	Jumps to the label if the 1st popped value is 1.

`:>label`

Aka:	Recursive call
Description:	Jumps to the label with the recursive mode.

`?:>label`

Description:	Jumps to the label with the recursive mode if the 1st popped value is 1.

`_name_`

Description:	Defines a skipper.

`_>name`

Description: Jumps to the skipper.

`_?>name`

Description: Jumps to the skipper if the 1st popped value is 1.

`wd-begin`

Description: Defines the word.

`wd-end`

Description: Ends defining the word.

Usage: `wd-end(n) name`

n ... Number to specify the minimum arguments. (Optional)

```
?(  ).
```

Description: If statement.

```
?!(  ).
```

Description: If not statement.

CHAPTER 10. THE RPL++ LANGUAGE: *Environment Variables*

There are 7 environment variables in RPL++.

Name	Type	Description
undef	undef	undef
notnum	notnum	notnum
#ARGS	Array (String)	Contains the passed arguments
#LINE	Number	Contains the line index of the program where the interpreter is running
#CMD	Number	Contains the word index of the program where the interpreter is running
#ROWS	Number	Contains the console rows
#COLUMNS	Number	Contains the console columns

CHAPTER 11. THE RPL++ LANGUAGE: *Errors*

There are 4 errors in RPL++.

Name	Description
InternalError	(Used in the interpreter)
StackUnderflow	Thrown when the stack doesn't have enough elements.
UnknownWord	Thrown when the word is unknown.
IncorrectType	(Unused)

RPL++

RPL++

RPL++