# The TSRP Programming Language

(Temporary Stack Reverse Programming)

---

`Warning: This documentation is still rather raw and was written in a few hours. Some of the descriptions might be ambiguous, and all the features might not be documented yet. There is a ton to document.`

TSRP is a stack-based anonymous functional programming language (with support for many number types) designed for both academic and demonstration purposes.

While languages like Lisp and Scheme use structured Polish Notation for their code, TSRP is designed to use a more chaotic and free Reverse Polish Notation, while sporting many of the same bragging points.

In TSRP, syntax is not tightly bound, as the language doesn't need to analyze what is valid.

```
{ 100 rand } 50 Repeat          { Generates 50 Random ints and Pushes them to the Stack } pop
{ + } 49 Repeat                 { Adds 49 times. } pop
```

---

## Reverse Polish Notation and The Stack

To understand the language, you must first understand stacks and reverse polish notation.

In a majority of cultures, math is typically written in the form "operand operator operand". This leads to complicated rules like PEMDAS where certain operations take priority over others.

In reverse polish notations, the operands are always written prior to the operator, so instead of being "10 + 20", it becomes "10 20 +". There are no ambiguous priority rules in RPN.

| Standard | RPN |
|---|---|
| 10 + 20 | 10 20 + |
| 10 + 20 * 30 | 10 20 30 * + |
| 10 * 20 + 10 | 10 20 * 10 + |

Reverse Polish Notation has been implemented in TSRP by pushing values onto a stack, and executing the operators on the top values on the stack.

Everything is pushed onto the stack in TSRP, including functions.

To remove variables from the stack that you no longer need, you must type "pop"

---

## Global Variables

In TSRP, there are two supported variable types. Local and Global variables. Let's focus on the global ones for now. Setting variables happens by peeking at the top value on the stack, and copying it to the variable space. It uses the "=VariableName" command.

```
10 =t
```

Would stick “10” into t. Note that it does not pop the variable off the stack. If you wish to do that, you must type “pop” after it, like so.

```
10 =t pop
```

To retrieve the value for later use, you can use the “$VariableName” command.

```
$t 20 +
```

---

## Functions, Ifs, Loops

In TSRP, all functions are first declared anonymously and pushed onto the stack. These functions also make it possible to implement loops and if statements. These functions can be named and run from the variable space, but can also be executed anonymously.

```
{ Anonymous Function Demonstration } pop
20
{
        10 +
} exec
```

Would produce an answer of 30. Note that running “exec” pops the function off the stack when run.

To create a named function, you simply push an anonymous function onto a stack, and assign it a variable.

```
{
        10 +
} =AddTen pop
```

To call “AddTen”, you have two options. You can either push it onto the stack and use a command like “exec” to run it (which pops it off the stack), or you can type the function name.

```
20 $AddTen exec

{ or } pop

20 AddTen
```

Pushing functions back onto the stack can be useful for passing them around as variables, or using them in loops/ifs.

To write an if, you must push a function onto the stack, then a boolean, and run the “if” command.

```
{
        30
}
$x 30 <
if
```

This function checks if x is less than 30, and if it is, push 30 onto the stack. Keep in mind that calling the “if” pops both the boolean and the function off the stack.

Else ifs are also supported. “{ true function } { false function } bool eif”

```
{
        _=x pop _=y pop
        {
                _$x
        }
        {
                _$y
        }
        _$x _$y < eif
} =Min pop
```

This is a simple “Min” function (with local variables) that takes two inputs and returns the lower value, keep in mind, however, that such a function is natively supported.

For loops, there are multiple options, you can use "Repeat"s, and "while"s.

Let’s start with a repeat.

```
0
$AddTen 20 Repeat
```

Runs the AddTen function 20 times. The top value on the stack will be 200.

You can also specify an incrementation local variable (discussed in the next session) with the repeat call.

```
0
{ _$c 1 + + } 20 Repeat:c
```

This calculates the summation of every value from 1 to 20 (the incrementer goes from 0 to the upper-bound, exclusive).

If you wanted to create a while loop, you need two functions. One will be the inner loop code, and the other will be what produces the boolean value that is checked.

```
0 =x pop
{ $x 1 + =x pop }
{ $x 10 < } while
```

This while loop would increment x until it equals 10.

---

## Local Variables

Locally scoped variables are similar to global ones, but they have a few interesting traits. They automatically erase themselves after they fall completely out of scope. However, they are designed to have a few interesting traits. (Note that this section may require reading of the next section).

```
{
        _=Hello _$Hello +
} =Double pop
```

Local Variables are also able to travel to the functions that are called by a function, and back to the functions that called them. This allows you to pass values “by reference” in a sense.

```
{
        10 _=John
        B
        $_John +
} =A pop

{
        20 _=John
} =B pop

A
```

Calling ‘A’ would produce a result of 30. Note that the use of this feature isn’t necessary, as you could return values simply by pushing them onto the stack.

If you wish to prevent the local variables from being modified by other functions, you can add a “hold” command prior to setting a local variable.

```
{
        hold
        10 _=John
        B
        _$John +
} =A pop

{
        20 _=John
} =B pop

A
```

A would produce 20.

However, keep in mind that the hold only applies to local variables set in the function AFTER the “hold” is set.

Local Variables also travel backwards up to calling functions. This is by design.

---

## Data Types

There are various data types supported.

| Types | Sub-Types / Description |
|---|---|
| Strings | N/A |
| Bools | N/A |
| Numbers | Bytes, Shorts, Ints, Longs, and Unsigned Variants. |
| Arrays | N/A |
| Custom | Extension Types that Allow Integration with the Scripting Language. |

Here are the methods supported for the “primitive” types.

Strings

| Type of Operation | Operations | Description |
|---|---|---|
| Conversion | array, carray, any of the number types below, function, string | “array” splits the string into strings of length one, and places it into an array. “carray” does the same, but converts each letter into a ushort. The number types convert the string to a number. Function turns the string into a function. string does nothing to it. |
| Conditional | ==, !=, contains | |
| Info | len, length | |
| Operational | +, split, trim | |

Numbers

| Type of Operation | Operations | Description and Notes |
|---|---|---|
| Arithmetic | +, -, *, /, %, ^ | Arithmetic Operations |
| Conditional | ==, !=, <, <=, >, >= | Conditional Operations |
| Bit-Wise | xor, or, and, \|, &, <<, >>, ~ | Not supported by floating point types |

| Type of Operation | Operations | Description and Notes |
| --- | --- | --- |
| Conversion | string, any of the number types in the list below | |

| Number Types |
| --- |
| byte |
| sbyte |
| short |
| ushort |
| int |
| uint |
| long |
| ulong |
| float |
| double |

To convert between any of these types, you can type it in as a command.

```
1 2 double /
```

Numbers are also dynamically promoted when there are mis-matched variable types. If the variables are of the same priority (unsigned vs signed), the deeper variable of the two is the one that they are converted to.

```
1 uint
1 int +
```

Would produce a 2 uint.

Bool

| Type of Operation | Operations | Description and Notes |
|---|---|---|
| Operational | !, not | |
| Conditional | ==, !=, &&, ||, ^, or, and, xor, nor, nand | |
| Conversion | string, any of the number types above. | |

StackArray

This type has abnormal behavior in comparison to the other primitives. Unlike the others, this one does not clone itself when it is copied to the vaiable space. It is also flagged as a “Custom” type, which means that even if it isn’t on the top of the stack, if it is the top custom type, it will execute commands.

```
[]
10 20 push push
```

Even though array_push isn’t a command for numbers, all commands that fail on the top data type will route to the nearest custom type, which is the array. So it pushes 20 and then 10 into itself.

Also, if you add a “*” to the end of any array command, if the array is the top value on the stack, it will pop itself from the stack when it completes its operation. If you add “**”, it will remove itself from the stack once it completes its operation, regardless of how deep in it is.

```
[]
20 30 40 push push push
{ other code } pop
{ other variables put onto the stack } pop
empty**
{ empties the array and pops it from the stack } pop
```

| Type of Operation | Operations | Description and Notes |
|---|---|---|
| Operational | empty, array_clear, array_pop, release, clone, reverse, push, array_push | empty/array_clear empties the array. array_pop pops a value from the array and sticks it onto the current stack. Release pops all the values in the array and puts it inside the stack. Clone actually creates a duplicate of the array, deep copying. |
| Conversion | chars2str, concatstr | chars2str takes all numbers in it, and converts it to a string. It ignores other data types. concatstr takes all strings inside of it, and adds them together. |

| Type of Operation | Operations | Description and Notes |
|---|---|---|
| Set/Get | at, set | at retrieves the element at a specified position in the array. set sets the element at a specified position in the array. |
| Info | size | |

However, this hardly tells the full story of what is possible with the language. We'll need to talk more about parser commands and how they interact with types like the StackArray.

---

## Parser Commands

The parser has a reasonably sized list of basic commands.

Oh, and there is a pre-processing step, so when you've seen

```
{ This is a comment } pop
```

That actually gets trimmed from the code.

| Commands | Description |
|---|---|
| swap | Swaps the two top elements. |
| dup | Duplicates the top element. |
| pop, del | Pops the top element. |
| $variableName, _$variableName | Pushes a variable onto the stack. |
| =variableName, _=variableName | Peeks at the stack and the top element to the variable space. |
| variableName | If the variable is a function, it will be executed. |
| if | Takes in a boolean and a function, executes the function if true. |
| while | takes in two functions. One for generating the boolean, one which is the inner loop code. Executes while true. |

| Commands | Description |
|---|---|
| repeat | takes in a number and a function. repeats function that number of times. You can specify an incrementation value which will be a local variable. |
| hold | tells the function to hold onto any local variables set after the hold is declared when calling other functions (pass by value). |
| exec, run, execute | if the top value on the stack is a function, it runs it. |
| import | Imports a custom type variable. This is for modules. |
| set_hold | Takes in a boolean and toggles the hold. In the current implementation, it does not undo the “hold” on values that have already been set after a hold. |
| @size | Pushes the current size of the stack onto the stack. |
| [] | push an array onto the stack |
| ##.# | push a double onto the stack. |
| ### | push an integer onto the stack. |
| “text here” | push a string onto the stack |
| true, false | push a boolean onto the stack. |
| ::variableName | if the variable is a StackArray, push the size of it onto the current stack. |
| :>variableName | if the variable is a StackArray, pop the top element of the current stack and push it into this array. |
| :<variableName | if the variable is a StackArray, pop the top element of the array, and push it to this current stack. |
| Enter, enter | **These features might be confusing.** If the top element of the stack is a StackArray, “enter” it, and use it as the current stack. All operations inside it will modify the array, but not affect the previous stack. |
| Exit, exit, Leave, leave | If you are currently inside another stack, these commands will take you back to the previous stack you were in. |
| [ | Enter a new stack. |
| ] | If you are currently in another stack, using this command will return you to your previous stack and push the current one onto your previous stack as a StackArray. |

Demonstrating “Switching” Stack Features

```
[
        { 100 rand 1 + } 100 Repeat
] =arrayOfIntegers pop

{ This creates an array with 100 Random integers ranging from 1 to 100. }
```

Let’s say you already have an array though, and you wanted to do some processing on it. You could push and pull from it, or you could enter the array and modify it.

```
{ In this example, you have an array of integers, and you want to add them all together to bec
ome one, then push that value onto the original stack. } pop

$arrayOfIntegers
Enter
{
        +
}
@size 1 - Repeat
Leave
release**
```

You can also push and pull from named arrays with ease using the “:>” and “:<” commands. Here is an alternative to the previous example.

```
0
{
        :<arrayOfIntegers +
}
::arrayOfIntegers Repeat
```

Also, you can use these tricks to be lazy about cleaning up your stack during processing and setting up variables, for example

```
[
        { 10 + } =AddTen
        { AddTen AddTen } =AddTwenty
        { AddTwenty AddTen } =AddThirty
        { AddTwenty AddTwenty } =AddForty
        { bunches of other declarations } pop
] pop
```

Now we need to discuss Extensions / Custom types.

---

## Custom Types, Extensions, and Imports

Now that features of the language have been discussed, it should be noted that the language is not capable of doing much (aside from calculation) without a few extensions. TSRP was designed to be easy to extend, simply by creating a new custom type. Two are built in with the language by

default (Console and System).

As mentioned earlier, custom types do not have to be the top variable on the stack to have commands executed on them. If a command fails on the top element, it will then send the command to the closest custom element.

You are also able to “import” custom types, so that they don’t even need to be on the stack for variables to be executed on them. Keep in mind that imported types have the lowest priority. (The import command pops the type off the stack)

---

## Console I/O

The console type allows you to accept input and give output through the console (somewhat obviously). Here are its commands:

| Commands | Description |
|---|---|
| WriteLine | takes a string and writes it to the console with a new line appended. |
| Write | takes a string and writes it to the console. |
| Read | Reads a line of user input (when they hit enter) |
| ReadChar | Reads in a single character the user types into the console. |

Here is an example Hello World Program *(finally, am I right?)*

```
$Console import

"Hello World" WriteLine
```

Now let’s go a bit more complicated, and take in user input.

```
$Console import

"Hello, "
Read
+ "!" + WriteLine
```

This code will output “Hello, <whatever the user types in>!”

Let’s make it a bit more complicated

```
$Console import
{ "Hello, " swap + "!" + WriteLine } =Greet pop

"Type in your name : " Write Read Greet
```

This will prompt the user for their name, then greet them appropriately.

What if you wanted to read in integers? Well, just convert the string!

```
$Console import

Read { Read something in } pop
int  { convert it to an int } pop

Read int  { Read in a second number } pop

+ { Add them } pop
string WriteLine { Convert it to a string and write it to the console } pop
```

That's really all there is to it. Command Line I/O is pretty easy with TSRP.

---

## File I/O

But what if you wanted to do file input and output?

Well, first you'd import system (or push it onto the stack), then you open a file.

| **File Supported Commands** | **Description** | **Status** |
|---|---|---|
| Read | Reads a single character | Supported |
| ReadLine | Reads a line. | Supported |
| Write | Writes a String | Supported |
| WriteLine | Writes a String with a new line character | Supported |
| CheckLine | Checks if there are more lines | Supported |
| ReadToEnd | Reads all the text to the end of the file | Supported |
| ReadWord | Reads a single word | Supported |
| Close | Closes the File | Supported |
| ReadByte | Reads a single byte (as a byte) | Experimental |
| ReadBytes | takes in a digit, and reads that many bytes to an array | Experimental |
| WriteByte | takes in a byte, and writes it to the file | Experimental |
| WriteBytes | Takes in an array of bytes, and writes it to the file | Experimental |

| | | |
|---|---|---|
| Check | Checks if there are more bytes to be read | Experimental |

`Experimental indicates that the code probably works, but hasn't been well-tested.`

```
$System import

"data.txt" open_file

{
        "Hello!" WriteLine
} 20 Repeat
Close pop
```

Output is fairly similar to the console, except you need to close the file when you're done.

`Todo: Add the ** feature to Closing the File`

If you had a bunch of lines with different numbers, and you wanted to add them, here is a program you could write.

```
$System import
$Console import


"data.txt" open_file
0
{
        ReadLine int +
}
{
        CheckLine
}
while
Close swap pop
string WriteLine { this would work since the file has been popped off and close } pop
```

Now let's say you wanted to do multiple I/Os at once, but didn't want to think about swapping variables and order and pushing and popping. Here is an area where the stack switching techniques can help.

```
$System import
[ $Console ] =Cons pop
[ "data.txt" open_file ] =File pop

{
        :>File
        $File Enter Write Exit pop
} =WriteF

{
        :>Cons
        $Cons Enter Write Exit pop
```

```
} =WriteC


{
        :>File
        $File Enter WriteLine Exit pop
} =WriteLineF

{
        :>Cons
        $Cons Enter WriteLine Exit pop
} =WriteLineC

{ pop } 4 Repeat


{
        "Hello" WriteLineC
        "Hello" WriteLineF
} =M

M M M
```

Of course, there are other options for writing those methods, too.

---

## Other System Features

System has support for these commands

| Commands | Description |
|---|---|
| open_file | Opens a file for input/output. |
| call_windows | Takes in a string and makes windows command prompt calls. |
| call_program | Takes in two strings, one is an executable name, the other is the arguments. It will execute the program with those arguments. |

You can use call_windows and call_program to open other programs.

Example

```
$System import
"program.jar" "-f James -l Dowthitt" call_program
```

---

## Misc Programs (Examples)

**Simple Fibonacci Term Computation** (input of 46 produces all values calculatable by an unsigned long)

```
$Console import

Read int =times

0 =counter
{ $counter 1 + =counter 1 - " " + Write } =WriteCounter

{ string WriteLine } =Print

WriteCounter 0 ulong =n_0 Print
WriteCounter 1 ulong =n_1 Print

{ WriteCounter $n_0 $n_1 + =n_0 Print } =A
{ WriteCounter $n_0 $n_1 + =n_1 Print } =B

{ A B } $times Repeat

{ pop }
{ @size 0 != } while
```

### Funny Complex Fibonacci Computation *(practically unreadable)*

```
$Console import

2
{ A B } Read int
{ dup " " + Write 1 + } =W
{ "" + WriteLine } =d
0 ulong =a
W 0 d
W 1 =b d
{ W $a $b + } =Q
{ Q =a d } =A
{ Q =b d } =B
{ pop } =e 6 Repeat Repeat e
```

Notice how the second Repeat near the bottom is loosely couples with the anonymous function { A B } and the number produced by “Read int” at the top.

### High Low Guessing Game

```
$Console import

False =Q pop

50 =Z rand 1 + =value pop

{
        "Guess a value between 1 and " $Z + " -> " + Write



        Read int =x pop
```

```
        {
                { "Higher" }
                $x $value < if

                { "Lower" }
                $x $value > if

                {
                        "Good Job!"
                        True =Q pop
                }
                $x $value == if

                WriteLine
        }
        {
                "Invalid Value, out of range." WriteLine
        }
        $x 1 >=
        $x $Z <=
        and
        eif



        "" WriteLine
}
{ $Q ! }
while


"The correct value was " $value + "." + WriteLine
```

Notice how in this example that the result string is pushed to the stack, then WriteLine is called, rather than it saying { “Higher” WriteLine }, { “Lower” WriteLine }. These are little programming tricks that are possible.

**Lottery Numbers (6 Numbers, 1-100, No Overlap)**

```
$Console import

[] =Nums pop
[ { _$c } 100 Repeat:c ]
{
        {
                _$q :>Nums
        }
        size rand _=s at
        _=q 1 >
        if

        -1 _$s set
```

```
}
{
        ::Nums 6 <
}
while
empty**
{ :<Nums " " + Write } 6 Repeat:c
```

### Square Numbers

```
$Console import
{ _$c 1 + 2 ^ "" + WriteLine } 20 Repeat:c
```

Prints out all square numbers from n=1 to n=20.