:)+++ Manual

Pierfrancesco Guida(29086388)
Iustin Gabriel Dinca(29469244)

May 2018

1 Introduction

:)+++ (pronounced smile plus plus plus) is a stream processing language whose operation is very similar to that of a Turing machine. You have N input tapes, One variable Tape(Length set at 100k variables, can be expanded by changing a value in the interpreter), and N output tapes(Up to 50 but can be expanded by changing a value in the interpreter, the output tapes will always be the same length as the input tapes, the interpreter makes sure of it by making certain adjustments that will not cause it to crash). Our language does not have types for variables, all variables store only integers but with some encoding you can obviously use whatever type if necessary. Due to the nature of the given tasks we believe it was unnecessary to add other types to variables. It supports arrays, but they have to be encoded in the variable tape in any way you wish, an example of this can be seen in the appendices in pr9.spl where we've encoded an array in the variable tape. It also supports pointers, but it is the programmer responsibility to encode them and handle them. You can, for example, access a variable at position N, where N is the value stored at position X. It also supports for and while loops which make it easy to iterate over arrays. It also has an until_end X loop which is a while loop that goes on while stream X has elements to read.

2 Syntax

2.1 Conditionals

:)+++ supports all conditionals(¡,¿,¡=,¿=,=,!=, and, or) and you can use parenthesis to build expressions. We will further refer to these as "COND" for the rest of the guide.

2.2 Arithmetic

:)+++ supports the following arithmetical operations: +,-,*,/,% and negation. Parenthesis are also supported for building complex expressions and if there is need for more complex mathematical operations such as power, these can be done by the programmer using loops and variables. We will further refer to arithmetic as "ARITH" in this guide.

2.3 Value

By "VALUE" we mean something that eventually becomes a simple integer but will be decided at run time. When something in the grammar requests a "VALUE" it can either be a simple int, an ARITH,or one of the following two commands:

2.3.1 getValue

Syntax: getValue (VALUE)
This command will return the value stored at position VALUE in the variable tape

2.3.2 read

Syntax: read INT
This command will return the next element in the input stream INT

2.4 Expression

An expression is a command that manipulates the tapes in some way, it does not return any value. A block of expressions is called a language, we will further refer to expressions as "EXP" and languages as "LANG". A language cannot be empty, but can contain the "pass" expression should you choose to create an empty block.

2.4.1 put

Syntax: put INT (VALUE)
put will append the desired VALUE to output stream INT

2.4.2 setValue

Syntax: setValue (VALUE1) (VALUE2)
setValue will change the content at position VALUE1 in the variable tape to be VALUE2

2.4.3 discard

Syntax: discard INT
discard simply skips the next element in stream INT

2.4.4 pass

Syntax: pass
pass literally does nothing, but is necessary in case of an empty LANG

2.4.5 while

Syntax: while (COND) LANG In our While loop COND is checked at the beginning of every cycle, and if it evaluates to true the LANG block is executed before checking COND again.

2.4.6 until_end

Syntax: until_end INT LANG
until_end is very similar to a while loop but the condition is that input stream INT still has elements that need to be read

2.4.7 for

Syntax: for (INT = VALUE1 ; COND ; VALUE2) LANG
The for loop has an index variable(INT) initialized at value VALUE1, it will operate while the condition COND evaluates to true and at the end of every loop the index will be summed to VALUE2, which is usually 1 or -1 but can be any VALUE(Note: Positive values are to be put WITHOUT sign). If COND evaluates to true LANG is executed.

2.4.8 if

Syntax: if COND LANG1 else LANG2
The if expression first evaluates COND, if it evaluates to true, LANG1 is executed, otherwise LANG2 is executed. Remember, no LANG can ever be empty but it can simply contain nothing but "pass", so even if you do not want an else block you still need to include elsepass.

3 Additional Features

3.1 Whitespace

Line breaks are not counted in the program, when it reaches the interpreter it's all interpreted as if it was one big line, so adding whitespace is completely at the discretion of the programmer.

3.2 Comments

Because of how whitespace is handled, there is no such thing as single line comments, however you can comment in the following way: // COMMENT \\

3.3 Turing Completeness

Due to the features we've added to this language and the way it operates, :)+++ is Turing complete and can handle any given task.

3.4 Arrays and Data Structures

While :)+++ does not natively support arrays and data structures, these can be encoded by the programmer using loops. For an example of this see the appendix about pr9.spl.

4 Summary of Syntax

4.1 Binding and Associativity

* / % bind tighter than + and -
Negation binds tighter than * / %
AND binds tighter than OR
All the previous operators associate LEFT

4.2 BNF Grammar

Start at LANG, see Syntax section for details

<LANG> ::= <EXP> <LANG> | <EXP>

<EXP> ::= put <INT> (<VALUE>) | setValue (<VALUE>) (<VALUE>) | discard <INT> | pass | until_end < INT> <LANG> | for (<INT> = <VALUE> ; <COND> ; <VALUE>) < LANG > | while (<COND>) <LANG> | if <COND> <LANG> else <LANG>

<COND> ::= <VALUE> < <VALUE> | <VALUE> > <VALUE> | <VALUE> <= <VALUE> | <VALUE> >= <VALUE> | <VALUE> = <VALUE> | <VALUE> ! = <VALUE> | <VALUE> and <VALUE> | <VALUE> or <VALUE> | (<COND>)

<VALUE> ::= <INT> | getValue (<VALUE>) | read <INT> | <ARITH>

<ARITH> ::= <VALUE> + <VALUE> | <VALUE> - <VALUE> | <VALUE> * <VALUE> | <VALUE> / <VALUE> | (<VALUE>) | -<VALUE> | <VALUE> % <VALUE>

<INT> :== <DIGIT> | <DIGIT> <INT>

<DIGIT> :== 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5 Appendix

5.1 pr1.spl

```
put 0 (0)
until_end 0 {
    put 0 (read 0)
    }

```

5.2 pr2.spl

```
until_end 0 {
    setValue (0) (read 0)
        put 0 (getValue (0))
    put 1 (getValue (0) )
}
```

5.3 pr3.spl

```
until_end 0 {
    put 0 (read 0 + (3 * read 1))
}
```

5.4 pr4.spl

```
setValue (0) (0)
until_end 0 {
    setValue (0) ((getValue (0)) + (read 0))
    put 0 (getValue (0))
}
```

5.5 pr5.spl

```
setValue (0) (0)
setValue (1) (0)
until_end 0 {
    setValue (2) (read 0)
    put 0 ((getValue (2)) + (getValue (1)) + (getValue (0)))
    setValue (3) (getValue (0))
    setValue (0) (getValue (1))
    setValue (1) ((getValue (2)) + (getValue (1)) + (getValue (3)))
    }
```

5.6 pr6.spl

```
setValue (0) (read 0)
put 0 (getValue (0))
put 1 (0)
until_end 0 {
    put 1 (getValue (0))
    setValue (0) (read 0)
    put 0 (getValue (0))
}

```

5.7 pr7.spl

```
until_end 0 {
    setValue (0) (read 0)
    setValue (1) (read 1)
    put 0 (getValue (0) - getValue (1))
    put 1 (getValue (0))
}
```

5.8 pr8.spl

```
setValue (0) (0)
until_end 0 {
   setValue (1) (read 0)
   put 0 (getValue (0) + getValue (1))
   setValue (0) (getValue (1))
}
```

5.9 pr9.spl

```
setValue (0) (3)
setValue (1) (0)
until_end 0 {
   setValue (getValue (0)) (read 0)
   setValue (0) (getValue (0)+1)
   for(2 = 3;getValue(2)<getValue(0); 1){
     setValue (1) (getValue (1) + getValue (getValue (2)))
 }
 put 0 (getValue (1))
}
```

5.10 pr10.spl

```
until_end 0 {
   setValue (0) (getValue (1))
   setValue (1) (getValue (2))
   setValue (2) (read 0)
   put 0 (getValue (2) + getValue (0))
   setValue (2) (getValue (2) + getValue (0))
}
```