Trout Language Manual

Getting Started

Trout is an interpreted dynamically typed imperative language which processes input one line at a time from **stdin** and produces lines of output to **stdout**. In order to run a trout program, invoke the trout interpreter with the filename of your program as an argument:

```
.\troutc myprogram.spl
```

Trout Basics

Trout's main primitive data type is the integer. With this data type alone, we can introduce the simplest Trout program which will print output:

```
1
```

When executed this program will take exactly one line of input and output exactly one line containing “1”, then terminate.

The first thing to notice is that we did not explicitly tell Trout to print or to take input. Trout handles this for us via **blocking IO.** Much like in other high-level languages, input in Trout is blocking - if we attempt to process a line of input, the program will block until that line has been input. However, Trout also features blocking output - specifically, the output blocks such that the total number of lines output can never be greater than the total number of lines input. This allows the Trout programmer to simply output lines as they are computed, without worrying about whether too many lines have been output. This form of blocking can also prevent unnecessary computation by introducing a form of *imperative laziness* which is discussed further in the appendices.

Trout waited for an input line because its output is blocking, but why does simply writing “1” cause it to output anything? Trout is **print-by-default**. This means that if a bare Trout statement evaluates to a value and that value is not assigned or otherwise consumed, it gets printed. The reasoning for this is that we want trout to be as simple and concise as possible, and the average trout program is unlikely to use bare expressions without wanting to print them, so this behaviour makes sense in general.stdin

Assignment and Computation

Values can be assigned to variables in Trout using the = operator. Variables are dynamically typed and variable scope is global. Variable names can be alphanumeric but must start with a letter, and they may not include any reserved words we will encounter later (specifically, **`if`** and **`IN`**).

Trout supports integer arithmetic using the following infix operators (listed in order of operator priority):

/	integer division
*	multiplication
+	addition
-	subtraction

If there is a need to override or clarify operator precedence, brackets may be used.

So for example, here is a program which outputs 20:

```
x = 5
y = 4
x * y
```

Note that the result of the last line is printed as it is a bare statement which evaluates to a value.

There is also a special null variable represented by an underscore. Assigning to the null variable simply discards the value, and reading from the null variable results in a runtime error. The intended use of this assignment is to suppress the printing of what would otherwise be a bare expression, and it is generally only needed in complex Trout programs. For example:

```
_ = 5
```

Has no effect, and most notably, **does not** print the 5.

Frames

As Trout processes input one line at a time, it makes sense to support a line of input as a data type. Trout implements this as a “frame”, which can be thought of essentially as a 1-dimensional array of integers.

In order to combine frames the comma operator can be used. As Trout is dynamically typed with implicit overloading, this also allows for the construction of frames. For example:

```
4 , 3 , 2 , 7
```

Will construct a frame equivalent to the input line

```
4 3 2 7
```

As each integer is cast to a 1-length frame, and these frames are then combined. Note that in the above example, the whitespace is optional.

Streams

Trout processes streams as 1-dimensional arrays of frames, so rather than seeing the input

```
1 2
4 3
2 2
```

as two parallel streams, Trout approaches is it as a stream of width 2. Similarly to frames, an operator (the ampersand) to append streams is provided. So, using integer, frame, and stream operators, the following expression

```
4 + 3 , 6 * 2 & 1 , 5 - 2 & 4 , 3
```

evaluates to a stream equivalent to the following input

```
7 12
1 3
4 3
```

With Trout’s print-by-default behaviour, the program containing the expression listed above would output each evaluated frame line-by-line, with each line being output in response to an input line. After all three lines are printed, the program terminates.

There are two special stream variables: **`IN`** which is the **`stdin`** input stream, and **#** which is an infinite 1-wide stream of 1s.

Control Flow

Trout supports if statements with the following structure:

```
if (<boolean expression>) <statement>
```

The statement is conditionally executed if the boolean expression is true. Boolean expressions can be composed using four operators:

- `==` evaluating equality
- `!` boolean NOT
- `|` boolean OR
- `.` boolean AND

As with integer arithmetic, brackets can also be used.

Trout also supports iterating over streams, which is the primary mechanism of actually handling input. The syntax for an iterator is

```
<stream> {
  [<statement>]
}
```

The statements within the iterator are executed for each frame in the stream. A number of special language features are available within the context of streams - firstly, the current frame can be accessed via index notation using **[n]** where n is the desired index, starting from 0. Secondly, the output of the iterator (i.e. the stream it evaluates to) can be written to using **<n>** (chevrons as opposed to square brackets). Lastly, the **break** statement (generally evaluated conditionally) is available within iterators, which immediately terminates the iterator and continues execution after it. As an example

```
q = s {
  <0> = [0] + [1]
}
```

Takes a (at least) 2 wide stream **`s`** and produces a stream (assigned to **`q`**) where each frame of **`q`** is the sum of the first two indices in each frame of **`s`**.

Here is a program which takes 3-wide user input and outputs the same input but with the middle number tripled:

```
IN { [0], 3 * [1], [2] }
```

Here we're using frame construction with the comma operator, and making use of the fact that the resulting statement is treated as a bare statement (print-by-default behaviour is unaffected by iterator context) and so is printed immediately.

Typing and Error Handling

Trout is dynamically typed with implicit casts, which is why it's possible to simply use the stream append operator on bare integers with no problems, or why **`(3)`** is considered a valid stream to iterate over (although the iterator will only run once of course).

Trout's general approach to error handling is not to error. That is to say, if something can be computed reasonably in Trout without errors, Trout will compute it even if it is not explicitly allowed according to the specification. For example, Trout can work with variable width input, even though this is not explicitly permitted by the spec. Where errors are unavoidable, Trout informs the user where the error occurred (parsing, runtime, or input parsing) and attempts to detail the specific error as best as possible. Users are encouraged to experiment with this.

Formatting

Statements in Trout are separated by newlines, so newlines have syntactic meaning. The current Trout interpreter should be capable of handling both UNIX and DOS style line endings, though for compatibility reasons UNIX style line endings are recommended.

Whitespace is used to separate input tokens in Trout, but neither whitespace nor indentation carries syntactic meaning as part of actual Trout programs. For readability, it is recommended that C-style indentation is used, but this is not required.

Trout supports single line comments starting with `//` and multi-line comments starting with `/*` and ending with `*/`

Appendices

Appendix 1: pr1.spl - pr10.spl

The programs given in this appendix are the code as submitted, but with additional comments and some non-syntactic whitespace in order to aid understanding.

pr1.spl

```
/*
    Prefixes a 0 to a stream of width 1 and length n
    Note that Trout’s blocking printing ensures this does not print
    more lines than are input.
*/

0 & IN
```

pr2.spl

```
/*
    Copy the stream horizontally.
*/
IN {[0], [0]}
```

pr3.spl

```
/*
    Add values in the stream
*/
IN { [0] + 3 * [1] }
```

pr4.spl

```
/*
    Each value in the stream is the sum so far.
*/
acc = 0
IN {
    acc = acc + [0]
    <0> = acc
}
```

pr5.spl

```
/*
    Do Fibonacci things.
*/

buf = 0

IN {

  // This is a simple pattern to remember all input values, by simply storing them
  // in a buffer.
  buf = [0] & buf

  fiba = 1
  fibb = 1

  acc = 0

  // We’re iterating over buf but we don’t want this process to print anything,
  // so we use the null assignment.
  _ = buf {

    acc = acc + ([0] * fiba)
    nextFib = fiba + fibb
    fiba = fibb
    fibb = nextFib

  }

  acc

}
```

pr6.spl

```
prev = 0

IN {
  [0], prev
  prev = [0]
}
```

pr7.spl

```
IN {
  [0] - [1], [0]
}
```

pr8.spl

```
prev = 0

IN {
  [0] + prev
  prev = [0]
}
```

pr9.spl

```
buf = 0

IN {
  buf = [0] & buf
  nat = 1
  acc = 0
  _ = buf {
    acc = acc + ([0] * nat)
    nat = nat + 1
  }
  acc
}
```

pr10.spl

```
buf1 = 0
buf2 = 0

IN {
  result = [0] + buf2
  buf2 = buf1
  buf1 = result
  result
}
```

Appendix 2: Prime Number Generation

As a demonstration of more complex Trout programming, we present an example of the Sieve of Eratosthenes implemented in Trout in order to generate prime numbers:

```
/************************************************************\

                    Prime Number Generator
                     Implemented in Trout

\************************************************************/

// We must give the sieve of eratosthenes the first prime.
curr = 2
primes = 2

# {
    divisorcount = 0
    _ = primes {
        quotient = curr / [0]
        if(quotient * [0] == curr) divisorcount = divisorcount + 1
    }
    if(divisorcount == 0) (0) {
        curr
        primes = primes & curr
    }
    curr = curr + 1
}
```

Appendix 3: Proof of Turing Completeness

In order to prove that Trout is Turing Complete, we have implemented a simple tape-based esoteric language known as brainf**k in Trout. A proof that brainf**k itself is turing complete may be found here: http://www.hevanet.com/cristofd/brainfuck/utm.b

```
/***********************************************************\

    Brainf**k - An esoteric turing machine-like interpreter.
    Implemented in Trout as a proof of Turing Completeness.

\***********************************************************/

// Variables
program = 0
programlength = 1
tapesize = 300 // Theoretically infinite ;) It loops in this implementation.

// Instructions

mvright = 0 // > - move the pointer right
mvleft = 1  // < - move the pointer left
incr = 2    // + - increment the cell
decr = 3    // - - decrement the cell
out = 4     // . - Output the cell
repl = 5    // , - Replace the cell with input
lbr = 6     // [ -  jump to the matching ] instruction if the current value is zero
rbr = 7     // ] - jump to the matching [ instruction if the current value is not zero


// Make a blank tape.
tape = 0
i = tapesize
# {
    tape = tape, 0
    i = i - 1
    if(i < 0) break
}

/*
    Main program loop.
    Executes one instruction per press of enter key.
*/
addr = 0
pc = 0

IN {
    program {instr = [pc]} // Get the instruction

    // Move Pointer Right
    if(instr == mvright) (0) {
        addr = addr + 1
        if(addr > tapesize - 1) addr = 0
    }

    // Move Pointer Left
    if(instr == mvleft) (0) {
        addr = addr - 1
        if(addr < 0) addr = tapesize - 1
    }

    // Increment Cell
    if(instr == incr) (0) {
        tape {<addr> = [addr] + 1}
    }

    // Decrement Cell
    if(instr == decr) (0) {
        tape {<addr> = [addr] - 1}
    }

    // Print
    if(instr == out) (0) {
        tape {<addr>}
    }

    // Read
    if(instr == repl) (0) {
        data = [0] // First value from stdin
        tape {<addr> = data}
    }

    // Left Bracket
    if(instr == lbr) (0) {
        tape {val = [addr]}
        if(val == 0) (0) {
            i = pc
            # { // NB: This can loop if there is no right bracket
                i = i + 1
                program {val = [i]}
                if(val == rbr) break
            }
            pc = i
        }
    }

    // Right Bracket
    if(instr == rbr) (0) {
        tape {val = [addr]}
        if(val != 0) (0) {
            i = pc
            # { // NB: This can loop if there is no left bracket
                i = i - 1
                program {val = [i]}
                if(val == lbr) break
            }
            pc = i
        }
    }

    pc = pc + 1
    if (pc > programlength - 1) break // Check for end of program
}
```

Appendix 4: Imperative Laziness

Trout’s blocking output implements a form of “imperative laziness”, which can prevent excess computation by using a tactic similar to lazy evaluation, despite the fact that Trout is an imperative language with side-effects.

Specifically, once an output line is buffered and waiting for an input line, computation blocks. As Trout supports arbitrary computation, this can be quite helpful. For instance, in the case where where the fourth line of output would cause non-terminating computation, but we only receive three lines of input and then an EOF, attempting to compute the fourth item is simply wasteful as the result will never be used.

This behaviour is reasonable precisely because the side-effects available to Trout are well defined, and consist exclusively of outputting lines. In a traditional imperative language, suspending computation arbitrarily can be problematic, as it is not trivially possible to determine which side-effects are intended, or when side effects should occur.

In Trout however, the only side-effect is outputting lines of integers, and the times when this is desirable are well defined (i.e. Trout should output if and only if the total number of input lines is greater than the total number of output lines). This provides the fundamental guarantee that the order and nature of side-effects cannot be modified by the suspension of computation, precisely because Trout has only one form of output operation.

In this way, a Trout program can be treated like a functional program in a monadic context of output lines. Indeed, the Haskell implementation of Trout effectively processes Trout programs in this manner (albeit using the broader IO monad for technical reasons).

In general we feel that writing Trout as an imperative language was the correct course of action, because for simple stream processing tasks, the simplicity gained by taking an imperative approach outweighs the advantages of a purely functional approach. However, it would be interesting to see if a language like Trout (or perhaps a further version of Trout) could be implemented as a purely functional language which, through implicit monadic contexts, appears to a typical end user to be imperative.