# katt64/troff-calculator

Stephanie Björk[1]

Written: November 21, 2017
Typeset: November 27, 2017

[1] Author’s pseudonym.

## 1. Introduction

*troff-calculator[1]* is an open-source programming project on Github (refer to the delayed text in the Bibliography for the link thither). It is a calculator capable of processing input expressions in infix notation and Reverse Polish notation[2] as typed by the human, and outputs the evaluation result onto the terminal or paper, wheresoever the user expects. Evaluation revolves around the general idea of a stack. All input pushed into the stack gets evaluated immediately whenever possible, a fact demonstrated clearly by the evaluation of Reverse Polish expressions.

The program along with all of its documentation was originally written and maintained by Stephanie Björk[3] when she was in high school. Since it is hosted on Github, anyone can fork or contribute to it whensoever they wish.

Generally, calculator programs are easy to make in general-purpose programming languages. As a matter of fact, most Computer Science exams like the GCSE Computer Science exam often instruct the programmer to write a calculator program using one of the main-stream programming languages: Python, Pascal, or Java to name a few. What sets this calculator program apart is not a unique feature in its functionality,[2] but rather the fact that the entire calculator program is purely written in plain Troff, a language designed to produce beautifully-typeset documents, and without any macro packages or outsourcing to other programming languages, not even user-friendly wrappers. This very fact is made crystal clear by Github's programming language analysis — 100% Roff.

Undoubtedly, Troff is a remarkably *terrible* programming language, albeit remarkably as versatile. This is not unsurprising, for Troff is a typesetting language, not a programming language. However, it is possible, as proven by this calculator program, to implement a few notable algorithms in Troff and coerce it into running them successfully. Although it works very well, the means to input and output data can be rather unconventional, mostly involving "line coding," and can be extremely user-unfriendly.

Nevertheless, this program serves as a demonstration of what can be done in Troff, which transcends its core values in typesetting. It was fun to write, yet many hurdles always come along the way every now and again. Since this has proven to be a challenge, things can get unpredictable. So, this program is provided to the public domain for educational purposes only, and it is not always guaranteed to work as a serious, fully-functional calculator although it does try to.

### 1.1. Coming from *rpn-calculator*

You might have noticed that this is not the first calculator of a similar kind in Troff. If so, you are right; it is not. This calculator is based on a predecessor, my previous *rpn-calculator*[4] also in Troff. This was rewritten and refactored from scratch to be generally better and more efficient than the predecessor. New features and improvements have been added, resulting in this current incarnation of the calculator. New features include:

- The ability to parse infix notation. The predecessor could only parse Reverse Polish notation, which meant that the user had to manually convert their expressions in infix notation to Reverse Polish notation.
- A practically unlimited input stack, so input expressions can be however long they have to be. The predecessor had an artificially limited input stack, which meant that the input expression could only consist of upto 26 tokens in Reverse Polish notation. It also meant that the input expression were not evaluate immediately, but were delayed until requested for. In this implementation, this limit has now been lifted and tokens get evaluated immediately whenever possible, making obsolete the request to evaluate input tokens at once.

[2] However, as far as I am concerned, it is the only actively-maintained calculator program that can parse Reverse Polish notation.

As of today, the predecessor is now officially intentionally abandonned and unmaintained, as this current implementation proves to be a lot better and cleaner. The predecessor is still kept on its own, original Github repository for historical and record-keeping purposes. All development efforts should be focused on this implementation, not on the predecessor anymore.

## 2. Capabilities

This calculator's capability is a superset of its predecessor's capabilities. It contains the same features, functionality, and naming conventions as in the predecessor, but also adds and improves upon many of those.

### 2.1. Input tokens stack

No longer do input tokens get imprisoned within a severely confined 26-slot "input tokens stack" and processed at once at the end of the program. This means that input expressions can be however long they must be, as it gets simplified and evaluated on-the-go. However, there is still a limit on how many operands can go unevaluated consecutively within the evaluation stack; that limit is 26. There is an internal and technical why the limit is exactly 26, and it has to do with the low-level inner-workings of the implementation of the stack. This point will be most prominent in the Troff source file of the stack algorithm.

To make this clear, if you use Reverse Polish notation and push 26 operands into the evaluation stack with no operators in sight to allow for evaluation, the calculator will refuse to accept more operands into its evaluation stack. This means that if you push one more operand to make a total of 27 operands pushed, you can kiss that operand goodbye, because it simply gets rejected. The program will give you no run-time warnings whatsoever about this, so consider this paragraph as your only forewarning. However, if after 26 operands have been pushed into the evaluation stack and an operation is found, the calculator program can now evaluate something in its evaluation stack, ergo freeing up 1 space in the stack. This means that you can push 26 operands to fill the stack, clear the stack by adding them all up with the addition operator, and push another 25, clear the stack by adding them all up, and push another 25, and ..., going on virtually indefinitely as long as the (intermediate) numerical result is machine-size.

### 2.2. Numerical computation

The calculator is capable of numerical evaluation, reading numerical input tokens, evaluating them, and outputting the result numerically. It is not capable of any form of symbolic computation found in LISP or other major Computer Algebra Systems (CAS). This means that there is a practical limit as to how large of a number the calculator can handle, just like there is a limit of how big an `int` type integer can be in C or JavaScript.

This calculator is capable of working with positive integers, 0, and negative integers about equally as well, just like most calculators. Unlike most conventional calculators, however, this calculator is not capable of handling floating-point arithmetic. That simply means that floats and decimals do not work, only signed integers including 0 do. If an evaluation must result in a decimal, like $11 \div 3$, the result is simply truncated with its decimal point stripped off (not rounded) to the greatest integer that is less than or equal to the actual result. In other words, $11 \div 3$ does not evaluate to $3.66\overline{6}$, but rather just $\lfloor 11 \div 3 \rfloor = 3$. This eliminates the hassle of implementing floating point arithmetic that could introduce potential roundoff errors as prominent in Fortran `REAL`s. If you need to work with decimals of limited precision, increase their order of magnitude and decrease it back to original after working with them.

Integers can be quite big, reaching well over $10^6$. However, it solely depends on how big Troff's numerical register (`.nr`) can store. I do not really know the exact number, for I have not found it in the User's manual, but I estimate it to be a lot more than $10^6$. :p

Since signed integers are the only supported numerical type, *operands* to the calculator solely consist of those such integers.

### 2.3. Supported Operators

The following list shows supported operator tokens and the character by which they are referred to when parsing them into the calculator. They are listed in increasing order of hyperoperations, not precedence in infix notation.

- Addition (`+`)
- Subtraction (`-`)
- Multiplication (`*`)
- Integer Division[3] (`/`)
- Modulo (`%`)

For example, if you would like to evaluate $15 \div 5 + 2 \times 3$ in infix notation, you will want to parse it into the program as `15 / 5 + 2 * 3`. If you would like to evaluate the same thing $15\ 5 \div 2\ 3 \times +$ in Reverse Polish notation, you will want to parse it into the program as `15 5 / 2 3 * +`.

Exponentiation and tetration are too complicated and thus are not supported.

Modulo is the remainder of a division. So, if you have 17 cookies and you want to share them equally among 3 friends[4], each of your 3 friends will get $17 \div 3 = 5$ cookies, and the remainder nobody can get equally is $17 \bmod 3 = 2$; because $17 \div 3 = 5$ remainder 2.

Since the order of operations is an important concern to infix notation but not Reverse Polish notation (hereafter called postfix notation because it is shorter), parentheses are also allowed for infix expressions:

- Opening parenthesis (
- Closing parenthesis )

They can be nested to however many orders you want, but you must remember to close them properly or you will get a parity warning. In postfix notation, parentheses are not necessary; therefore, they are not allowed as operator tokens.

### 2.4. Order of operations in infix notation

One of the hardest parts of writing this calculator is implementing the algorithm to parse Infix notation in such a way that operations get done in the correct, standard order as taught in middle school. This matter does not concern users of Postfix notation, ergo they do not need to bat an eye at this subsection. Nonetheless, it has been done, and the order of operations are hard-coded and trained within the Infix notation processor itself.

The order of operations programmed into the processor is a modified version of **PEMDAS**: Parenthesis, Exponentiation, Multiplication, Division, Addition, Subtraction, where operator precedence decreases as a function of each word in the acronym. However, since exponentiation is not supported but modulo is, the order of operations is actually something like **PMDMAS**: Parenthesis, Multiplication, Division, Modulo, Addition, Subtraction. All infix operators are left-associative.

In the infix processor, all operators are assigned values that denote the precedence of the operators. These values are used internally by the processor so it can collate precendences numerically. The following is a list of operators and their precedence values, ordered from highest to lowest precedence. Precedence values have a positively linear correlation with actual precendence.

[3] Since there is no floating point capability, only integer division is supported. This applies to other operations whose operands are non-integers.

[4] Both operands are co-prime to make this clear.

| | | |
|---|---|---|
| Opening parenthesis | 6 | ( |
| Multiplication | 5 | × |
| Division | 4 | ÷ |
| Modulo | 3 | mod |
| Addition | 2 | + |
| Subtraction | 1 | − |

Note that the order of operations done by the Infix processor respects the **PMDMAS** order *exactly* as is. What this means is that the infix notation, $30 \div 3 \times 2$, gets evaluated as $30 \div (3 \times 2) = 30 \div 6 = 5$; *not* $(30 \div 3) \times 2 = 10 \times 2 = 20$. So, to get $(30 \div 3) \times 2$, you must either parse it as: $30 \times 2 \div 3$, or use parentheses explicitly: $(30 \div 3) \times 2$. Converting operator symbols into machine-parsable ASCII is left as an exercise for the reader.

## 3. Input expressions

I have been using the words: *expressions* (input expressions), *tokens*, *operators*, and *operands*, quite a lot to mean their respective and specific things. These are the fundamental building pieces for input expressions into the calculator. To give a picture of how they rank up relative to each other and what they mean, a formal definition is given in a slightly modified version of the Backus-Naur form in figure 1. Note that parentheses are *illegal* in postfix notation expressions, so they only ever matter if you are using infix notation. Remove the non-terminals and terminals relating to parentheses when working with postfix notation.

From this formal definition, we can derive an ‘infinite’ list of possible input expressions that this calculator should be able to handle, although they must be formatted correctly and sensibly in infix or postfix notation.

As examples, $((3 + 2) \times (5 - 3)) \div (2 + 1)$ is correct infix notation and follows the formal definition, and $3\ 2 + 5\ 3 - \times\ 2\ 1 + \div$ is correct posfix notation and follows the formal definition excluding the terminals and non-terminals concerning parentheses. However, $((3 + 2) \times ((5 - 3) \div (2 + 1)$ is incorrect infix notation, even though it follows the formal definition: parentheses parity is postive $1 > 0$, i.e. unclosed parentheses. $3\ 2.0 + 5e6\ 3 - \times\ 2\ 0.9 + \div$ is correct postfix notation, but does not follow the formal definition: no decimal points or exponential forms are allowed in the formal grammar.

To prepare the input expression for the calculator to evaluate, you will certainly not be parsing them as expressions separated by spaces as you are used to, but rather “surrounded” by certain requests to push them and automatically evaluate on-the-fly as possible. However, the correspondence between the usual input expression format you are used to and the input expression you have to specifically format for the calculator still remains 1:1, without any insertions or editing required. Details on how to prepare your input expressions for the calculator are described in the next section on Operation.

```
<digits> ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’
<unary> ::= ’-’
<operand> ::= <unary>? <digits>+
<operator> ::= ’+’ | ’-’ | ’*’ | ’/’ | ’%’
<parenthesis> ::= ’(’ | ’)’
<token> ::= <operand>
<token> ::= <operator>
<token> ::= <parenthesis>
<expression> ::= <token>+
```

Figure 1. Backus-Naur form for input expressions to the calculator

## 4. Operation

This section describes proper usage of the program in order to guarantee desired results as much as possible. The result of evaluation is highly dependent on the correctness of the input expression in the first place. This section will not deal with that, but rather deal with the more technical aspects in gathering all resources required, preparing input, processing it, and dealing with output, grouped into their respective sections.

Begining from § 4.2. Input, I will be using an input file throughout the sections thereafter as a test case, so that the examples are followable and consistent.

### 4.1. Prerequisites

To guarantee a successful run, a few source files are needed and must be fetched from the project's Github repository as required using `git clone`. When fetching those files, store them into one same directory and preserve their directory hierarchy. Those files are listed below.

`infix.roff`

This is the infix notation processor. It uses the shunting-yard algorithm[5] to internally convert the infix expression into a postfix one which then gets evaluated internally as postfix. In this way, it serves more as a human-friendly wrapper to the postfix notation processor (`rpn.roff`) by automatically converting infix expressions to postfix rather than actually working with the infix expression notation directly. Consequently, not only does it support the 5 operators as does the postfix processor, it also supports parentheses, for it is often required by infix expressions. It contains subroutines that accept tokens from an infix expression one-by-one and evaluate them almost on-the-fly, and clear the operator stack (equivalent to pressing the $\boxed{=}$ button on a calculator) and output the result. All user-accessible subroutines are named beginning with `IN`; for example, the request to push tokens from the input expression is called with **`.IN`**`PUSH`. You will only ever need this processor if your input expressions are in infix notation and you do not want to manually convert them to postfix yourself. *This processor is experimental, and may yield inaccurate results.*

`rpn.roff`

This is the Reverse Polish notation or Postfix notation processor. It uses the generic algorithm to evaluate postfix expressions immediately whenever possible as input tokens get read, and it does so using the idea of a stack. It contains subroutines that accept tokens from a postfix expression one-by-one and evaluate them on-the-fly whenever possible, and print the result when there are no more tokens to input. All user-accessible subroutines are named beginning with `RPN`; for example, the request to push tokens from the input expression is called with **`.RPN`**`PUSH`. Since postfix notation is the heart of all the parsing in this calculator, it is a low-level way to use the calculator itself.

`stacks/alpha.roff`

This is a minimalist implementation of a stack codenamed "Alpha" ($\alpha$). It contains basic subroutines to push items into the stack and pop them off.[5] Subroutines and stack slots are referred to internally beginning with an `A`; so the stack counter[6] is called **`A`**`c`, the 3rd item of the stack is called **`A`**`C`, and the subroutine to push is called **`A`**`PUSH`.

`stacks/beta.roff`

This is a minimalist implementation of a stack codenamed "Beta" ($\beta$). It contains basic subroutines to push items into the stack and pop them off. Subroutines and stack slots are referred to internally beginning with a `B`; so the stack counter is called **`B`**`c`, the 3rd item of the stack is called **`B`**`C`, and the subroutine to push is called **`B`**`PUSH`.

[5] Unlike most stacks, this stack implementation's pop subroutine cannot return the value of the item it popped off. So, the top of the stack must be processed internally, and only then is the pop subroutine used to delete the top item of the stack.

[6] A stack counter counts the current number of items in the stack. Incidentially, this means that it can represent the slot ID of the last item in the stack, just like array indices.

I feel the extreme need to mention the most useless file in the repository that you will ***never ever*** need: `LICENSE`. It does not take much for a 17-year-old girl like me to realize that life is just too short to read shit like that. Please just give no fucks about it and throw it in the fucking trash if at all possible.

Source files may depend on each other. The following is a list of dependencies for each of the source files. On the left column is the source file's name and on the right are the dependencies it has separated by spaces.

```
infix.roff          rpn.roff stacks/beta.roff
rpn.roff            stacks/alpha.roff
```

Dependencies are recursive, so try to look at it as a recursive tree. Stacks are minimal and therefore do not have any dependencies. Not all dependencies are needed; for instance, if you do not require Infix processing, then you do not need `infix.roff`, which then means you do not need `stacks/beta.roff`. All this information can help you determine what you really need to fetch from the repository, as not all files are needed.

Source files of a programming language are nothing with the compiler or REPL for it. The Troff source files need a Troff compiler. Most distributions of GNU/Linux and Cygwin will include GNU Troff, a free Troff compiler. The reason for this is because most manual pages in Linux are typeset using Nroff, which uses the same set of tools used by this program. This makes this calculator program extremely portable. However, most BSDs like OpenBSD do not have GNU Troff installed by default; if you do not have GNU Troff installed, you will need to consult your operating supervisor's manual on how to install it.

## 4.2. Input

The calculator accepts input expressions and evaluates them appropriately. It understands two popular notations: infix notation and postfix notation. The choice of which notation to use will directly determine which input processor must be used. For infix expressions, the tokens are parsed using the `infix.roff` processor, and for postfix they're parsed using `rpn.roff` processor.

As a quick start to using the calculator, two files have been provided in the repository's root directory: `test.roff` and `testrpn.roff` as example input prepared for evaluation with the infix processor and postfix processor respectively. Both of the test case files have their own specific over-the-counter documentation, so one need no know Troff to start using the calculator.

To give further examples, henceforth consider the input expression $(4 + 5 - 2) \times 3 + 2$ as our test case. Since one can format the expression in either infix or postfix, this subsection will be broken down into 2 subsubsections to address each of them.

### 4.2.1. Infix notation

Consider the input expression $(4 + 5 - 2) \times 3 + 2$. The expression is already in infix notation, so one need not convert any further to any "fancy" notations when preparing it for the infix processor. What one needs to do, however, is the easy task of breaking up the expression into individual tokens. As illustrated by the formal grammar in figure 1, tokens can be operands, operators, or parentheses. This is easy for a human to do by eye because tokens in an expression are simply delimited by whitespace in most general cases. Therefore, breaking down the input expression, the tokens are:

| | |
|---|---|
| Parenthesis | ( |
| Operand | 4 |
| Operator | + |
| Operand | 5 |
| Operator | − |
| Operand | 2 |
| Parenthesis | ) |
| Operator | × |
| Operand | 3 |

| | |
|---|---|
| Operator | + |
| Operand | 2 |

Notice the direct 1:1 correspondence of our token list above with the original input expression. Each token is now separated using a new line instead of whitespaces and written vertically instead of horizontally as the input expression was. Since the calculator can only accept tokens one-by-one and each command (request) to register a token into the calculator must be on a line of its own, we will have to parse tokens vertically just like this. As far as I know, Troff simply does not have the capability to split a string macro using whitespaces as delimiters and store each extracted token automatically: you are just left with no choice but to do it yourself.

Remember that you must convert each operator to their ASCII equivalents; since you cannot trivially type symbols like × or ÷ on a US ASCII keyboard, the calculator only understands ASCII operator symbols, and nobody in their right minds is using a space-cadet keyboard nowadays. So, replace× with an asterisk '*' and ÷ with a forward slash '/' when looking at each operator in the token list. For more information on the ASCII equivalents for operators, see § 2.3. Supported Operators.

Now that the tokens have been extracted and operators converted, all that remains is to intersperse them with requests to register each token into the calculator, and top it all off with a request to evaluate them all at once. The following is an example input file for the input expression. Notice the direct 1:1 correspondence with the token list. Requests start with a `.` (dot) on the first column of the line[7], followed by a name, followed by a space, and then a list of argument(s). Comments begin with `\"`; any text after which is completely ignored by the Troff compiler until a newline is found.

```
.so infix.roff \" Source the infix processor
.INPUSH ( \"      Register operator (
.INPUSH 4 \"      Register operand 4
.INPUSH + \"      Register operator +
.INPUSH 5 \"      Register operand 5
.INPUSH - \"      Register operator −
.INPUSH 2 \"      Register operand 2
.INPUSH ) \"      Register operator )
.INPUSH * \"      Register operator ×
.INPUSH 3 \"      Register operand 3
.INPUSH + \"      Register operator +
.INPUSH 2 \"      Register operand 2
.INEVAL \"        Evaluate and print the result.
```

The request `.so infix.roff` tells Troff to import infix processing capabilities from the file `infix.roff`. It only needs to be called once at the beginning of your input. Otherwise, there are two available requests at your disposal:

`.INPUSH t`
: Accepts 1 required argument: `t`. This request registers the infix token `t` into the calculator. `t` must be an integer operand or one of the supported operators (see § 2.3. Supported Operators) or a(n) opening/closing parenthesis. Use it as you wish, but your antecedent input must be sensible and correct.

`.INEVAL`
: Accepts no arguments. This request is equivalent to pressing the $\boxed{=}$ button on a calculator. It evaluates the input expression registered into the calculator and outputs the evaluation result to the terminal where the user can see.

[7] By the way, requests can also start with a `'` (single quote). Since the requests you will be using are generally user-defined, you do not need to worry about the differences.

### 4.2.2. Postfix notation

Consider the input expression (4 + 5 − 2) × 3 + 2. This expression has been given in infix notation. **Not good!** We must convert it to postfix. If you do not want to convert such an expression by hand, feel free to use the infix processor demonstrated in the previous section; it will internally convert your infix expression into postfix automatically. However, if you can bother yourself to convert such an expression to postfix, this section is for you.

The input converted to postfix notation becomes 4 5 + 2 − 3 × 2 +. Once such a conversion is done, all we need to do is to extract the tokens as before. The following is a list of the tokens in the postfix expression.

| | |
|---|---|
| Operand | 4 |
| Operand | 5 |
| Operator | + |
| Operand | 2 |
| Operator | − |
| Operand | 3 |
| Operator | × |
| Operand | 2 |
| Operator | + |

The beauty of postfix notation is that the expression is a lot shorter than its infix counterpart and it is bereft of parentheses. There are now less tokens to pass and it is considerably faster than infix notation for the machine to run, although most of us are not used to seeing or doing mathematics this way.

All that is left for the user is to convert operators to their ASCII equivalents and intersperse the appropriately-converted tokens with requests to register tokens into the calculator, and top it all off with one request to output the result to the terminal. The following is an example input file for the calculator. The syntax is exactly the same as before.

```
.so rpn.roff \" Source postfix processor.
.RPNPUSH 4 \"   Push operand 4 into evaluation stack.
.RPNPUSH 5 \"   Push operand 5 into evaluation stack.
.RPNPUSH + \"   Pop 2 items from stack, + together, push result back.
.RPNPUSH 2 \"   Push operand 2 into stack.
.RPNPUSH - \"   Pop 2 from stack, − together, push result back.
.RPNPUSH 3 \"   Push operand 3 into stack.
.RPNPUSH * \"   Pop 2 from stack, × together, push result back.
.RPNPUSH 2 \"   Push operand 2 into stack.
.RPNPUSH + \"   Pop 2 from stack, + together, push result back.
.RPNPRINT  \"   Print last item in stack to terminal.
```

The request `.so rpn.roff` tells Troff to import postfix processing capabilities from the source file `rpn.roff`. This request need only be called once at the beginning of your input. There are similar requests to register postfix tokens and output the result, but they behave drastically differently from the requests for infix notation and thus do not have the same names. The following 2 requests are available at your disposal:

`.RPNPUSH t`

Accepts 1 required argument: `t`. This request registers the postfix token `t` into the calculator. `t` must be an operand in the form of a signed integer or an operator listed in § 2.3. Supported Operators. If `t` is an operand, `t` gets pushed into the 26-slot evaluation stack, but is silently discared if the stack is full. If`t` is an operator , the evaluation stack is popped twice, the two popped items get evaluated according to the operator, and the result is pushed back onto the evaluation stack.

`.RPNPRINT`

Accepts no arguments. This request outputs to the terminal the***first not last*** item of the stack with the assumption that it is the final result of the evaluation. This assumption is usually correct if the evaluation stack counter is 1; i.e. there are no orphan operands left unevaluated on stack, a case that can only happen if the input expression is malformed.

As you can see, postfix notation makes it extremely clear that when you register your tokens into the calculator, they get evaluated immediately whenever possible; whereas infix notation is a little more complicated and less straightforward, depending on various circumstances. This is, because of the simplicity of postfix notation in the eyes of a computer. Consequently, because of this simplicity, postfix is the heart-and-soul, low-level interface of this calculator,[8] as all input expressions, be they in infix or postfix, get converted if necessary and evaluated as postfix by this calculator.

## 4.3. Processing

After receiving input, the calculator uses builtin algorithms trained into its code written in Troff. No part of the logic is outsourced to other programming languages during processing, as everything is done in pure Troff, following the philosophy of the calculator program. The matter of which algorithms are used by the calculator, how it runs, and how long it takes to run, is directly correlated to the given input expression and the notation in which it is set (infix or postfix). This subsection explains how to coerce the Troff compiler (GNU Troff) into running the necessary algorithms to evaluate the input expression given by an example input described in the same input file from the previous section. The algorithms, how they work, and how they are run will also be detailed as well. Since the calculator has algorithms to process infix and postfix expressions, this subsection will be broken down into two subsubsections, describing the evaluation of each respective notation systems.

Technically speaking, all processing done by the calculator is based on the notion of a stack. In fact, all algorithms used by the calculator, big or small, are stack-based. Therefore, builtin to the calculator are sufficient algorithms and subroutines covering them that implement a minimalist stack that works sufficiently well, just enough to fulfill this calculator's purpose. The algorithms directly pertaining to the stacks, however, are relatively minute and trivial comparing to the other equally-important albeit more-complex algorithms involved in expression evaluation. Thus, only occasionally will the simpler of algorithms be mentioned here, for the author believes that the best explanation for such a simple implementation is to look at the source code, which is no more than 20 lines long.

### 4.3.1. A word on stacks

The way minimalist stacks are implemented in this calculator is that they each take up 26 numerical registers in Troff. There are failsafes builtin to verify that stacks are not overpushed or overpopped beyond 26 or 0 respectively. However, there are no failsafes builtin to prevent popping or overwriting to a numerical register that has already been used by something else. In fact, the only way this stack implementation knows about items that exist in the stack is by looking at the cardinality of the stack — how many items are in the stack. It maintains the number by incrementing/decrementing the stack counter register when a push/pop routine is called respectively. Thus, the cardinality of the stack is stored in the stack counter register.

Now, given the premises that:

- Numerical registers can only be 2 characters long.
- A numerical register's value can be represented lexicographically. The lexicographical representation is A-Z (one character) if the register's value is between 1-26 inclusive, but is AA-ZZ (two characters) if the register's value is between 27-676 inclusive, and so on.

[8] This utterly depends on how you would define "low-level" and how deep you are willing to go. Yes, within its scope of control, the calculator handles tokens in postfix notation at the core, but the actual instruction to evaluate tokens in Troff is done using Troff's default infix notation for numerical registers. The Troff compiler was likely written in C, which uses infix notation, but the lowest-level assembly code for Troff could actually be doing arithmetic in a similar way to postfix notation: push two values into two registers, add said values together and store them in another register.

The format for numerical registers artificially allocated by the stack implementation can only follow the form *xy*, where *x* is the uppercase character abbreviation of the stack's name (Alpha or Beta), which means one character is already used, and *y* is the item identification code for the stack in uppercase, which leaves 0 characters left. If we had allowed identification codes like `AA`, there would have been 3 characters in the numerical register, which is not allowed.

This ultimately means that stacks can only have 26 slots maximum, and it is why there is a limitation of 26 slots imposed on the evaluation stack.

Take the alpha stack ($\alpha$) as an example. Items in the stack are stored in numerical registers begining with an uppercase `A` for Alpha and the cardinality of the stack is stored in the register `Ac` for Alpha cardinality. Items in the Alpha stack are stored in numerical registers labelled lexicographically from `AA` to `AZ`. If we had allowed 27 slots in the stack, we would have had to refer to the last item as `AAA`, which would have been an invalid register name for Troff.

### 4.3.2. Infix notation

Expressions set in infix notation are processed with the infix processor `infix.roff`. The sole algorithm responsible for infix processing is the *shunting-yard algorithm*, named as such because its behaviour is similar to that of a railroad shunting-yard.[9]

This implementation of the shunting-yard algorithm processes individual tokens from an input expression set in infix and converts them to an equivalent expression set in postfix. This conversion is necessary as all evaluation is done by the calculator in postfix, not infix. Furthermore, processing tokens in postfix notation is necessary, as it is very easy to build programs that parse and work almost directly on postfix notation than it is with infix. Low-powered machinery can be powerful and versatile because they work with postfix, because it is very easy for them to work with. Thus, all processing in this calculator is done in its lowest-level form, and that form is postfix notation. The simplicity of postfix parsing can be seen very clearly: there are a lot more lines of code in `infix.roff` than there are in `rpn.roff` even though `infix.roff` does outsource some of its logic to `rpn.roff`!

Since information on which operations to carry out first is ambiguous in infix expressions, the algorithm assumes the **PMDMAS** order of operations (see § 2.4. Order of operations in infix notation). Any part of the default order can be overridden by using parentheses within the expression. As shown by the formal grammar in figure 1, parentheses count as valid tokens, but they are only truly valid if and only if their parity at the end of the expression is 0, i.e. they are properly closed. Parentheses are supported by the infix processor only, not the postfix processor.

Input tokens are registered into the calculator program using the user-accessible request `.INPUSH`, dubbed from *INFIX PUSH*. Tokens are registered, but not necessarily pushed into any specific stack. When some tokens are registered, they actually immediately instruct the calculator to perform evaluation of a stack, thereby popping more than pushing. In nowhere are those such tokens saved within the calculator. However, when some tokens are registered, they do cause themselves to be pushed into an appropriate stack, depending on what type of token they are, and no stacks are popped.

By defintion of the algorithm, the infix processor uses two stacks: stack $\alpha$ as the output stack and stack $\beta$ as the operator stack. The $\alpha$ stack is not operated on directly by the processor, but rather through the help of the postfix processor. The $\beta$ stack is in complete control of the processor however. The $\alpha$ stack holds the postfix operands which are output by the infix processor; when infix processing is done or whenever possible, the output operands are pushed into this stack. The $\beta$ stack holds the infix operator tokens which await to be registered into the postfix processor automatically whensoever the algorithm sees fit, or when `.INEVAL` is issued. Not only does `.INEVAL` print the evaluation to the terminal, but it also instructs the calculator that the end of the expression is reached; the next paragraph explains `.INEVAL` in more detail. While operand tokens are pushed into the $\alpha$ stack, operators are not. Operand and operator

[9] I, the programmer, have never actually seen a railroad shunting-yard before, but okey...

tokens output by this algorithm are ***registered*** into the postfix processor, not directly pushed into the $\alpha$ stack.

In a nutshell, the `.INEVAL` request, dubbed from *INFIX EVALUATE*, instructs the calculator that the expression has ended and to print the result on screen. Actually, that is not a complete explanation. Infix expressions cannot feasibly be evaluated on-the-fly as can postfix expressions. On some ocassions, there can be operators left over in the $\beta$ stack, and consequently operands left waiting to be evaluated in the $\alpha$ stack. The only way for the calculator to know to expect no more infix tokens and to evaluate and print the result for once is for the user to explicitly say that the infix expression has come to an end. On most conventional calculators, this is done by pressing the $\boxed{=}$ button. The `.INEVAL` request is analogous to that; it empties the $\beta$ stack and registers (dumps) all the operators from there into the postfix processor in the process of popping and registering one at a time. This gives the opportunity for the postfix processor to evaluate remaining operands in the $\alpha$ stack. If the expression be formatted well, the postfix processor shall pop the $\alpha$ stack just sufficiently enough to leave one final result behind in that stack. This remaining item in the stack is the result of the entire evaluation, which is then printed by `.INEVAL`.

Consider the same input expression from the last section: $(4 + 5 - 2) \times 3 + 2$. Assuming the expression has been broken down into tokens and the tokens are prepared in an input file to the infix processor like the one in § 4.2. Input§1. Infix notation, the following pseudo-description shows how the algorithm runs.

(1) *The 1st token is* `(` *— an opening parenthesis.* The token gets registered into the infix processor. Since it is an opening parenthesis, it has the highest precedence of all the operators it understands. Thus, it gets assigned the highest precedence value of 6. This gets pushed into the $\beta$ stack immediately. Opening parentheses metaphorically behave as a shield. In the $\beta$ stack, it essentially creates a sandboxed environment for subsequent operators after it, whilst protecting existing operators below it; when it has done its job, it leaves gracefully, leaving all other operators unscaved. That is why we say, "He protecc, he attacc, but most importantly... he `BCLEARBRACC`."

| | |
|---|---|
| $\alpha$ stack: | |
| $\beta$ stack: | ( |

(2) *The 2nd token is* `4` *— the operand 4.* Since this tok en is a numerical operand, it simply gets registered into the postfix processor, which pushes it into the $\alpha$ stack.

| | |
|---|---|
| $\alpha$ stack: | 4 |
| $\beta$ stack: | ( |

(3) *The 3rd token is* + *— the operator + (addition).* The operator is assigned the precedence of 2 and gets pushed into the $\beta$ stack directly without further ado, because there are no operators already in the $\beta$ stack (except for the parenthesis).

| | |
|---|---|
| $\alpha$ stack: | 4 |
| $\beta$ stack: | ( + |

(4) *The 4th token is* `5` *— the operand 5.* Since this tok en is a numerical operand, it simply gets registered into the postfix processor, which pushes it into the $\alpha$ stack.

| | |
|---|---|
| $\alpha$ stack: | 4 5 |
| $\beta$ stack: | ( + |

(5) *The 5th token is* − *— the operator − (subtraction).* The operator is assigned the precedence of 1. Unlike the + operator, it does not get pushed into the $\beta$ stack immediately. Since the+ operator has a higher precedence (2) than the − operator (1), the + operator has to leave the $\beta$ stack and get registered into the postfix processor. Only then can the− operator be pushed into the $\beta$ stack to take its place. Consequently, as the + operator is registered into the postfix processor, the two operands now in the $\alpha$ stack get evaluated with its result pushed back into the $\alpha$ stack by the postfix processor.

| | |
|---|---|
| $\alpha$ stack: | 9 |
| $\beta$ stack: | ( − |

(6) *The 6th token is* `2` *— the operand 2.* Since this tok en is a numerical operand, it simply gets registered into the postfix processor, which pushes it into the $\alpha$ stack.

| | |
|---|---|
| $\alpha$ stack: | 9 2 |
| $\beta$ stack: | ( − |

(7) *The 7th token is* `)` *— a closing parenthesis.* The closing parenthesis causes all operators above the first opening parenthesis in the $\beta$ stack to be cleared and registered into the postfix processor; the slaughter of operators continues until the first opening parenthesis is encountered, at which the onslaughts stop and the first same opening parenthesis it encounters is popped from the $\beta$ stack. In this specific case, the only operator to get registered into the postfix processor is the − operator from the 5th step, which then causes the postfix processor to evaluate tokens in the $\alpha$ stack like in the 5th step. There are now no operator tokens in the $\beta$ stack thus far. *Unlike opening parentheses, closing parentheses are* ***never*** *pushed into the $\beta$ stack or considered operators.*

| | |
|---|---|
| $\alpha$ stack: | 7 |
| $\beta$ stack: | |

(8) *The 8th token is × — the operator × (multiplication).* The operator is assinged the precedence value of 5, the second highest from the opening parenthesis. Since there are absolutely no operators already in the $\beta$ stack, this operator gets pushed immediately without further ado.

| | |
|---|---|
| $\alpha$ stack: | 7 |
| $\beta$ stack: | × |

(9) *The 9th token is* `3` *— the operand 3.* The operand immediately gets re gistered into the postfix processor, which pushes it into the $\alpha$ stack.

| | |
|---|---|
| $\alpha$ stack: | 7 3 |
| $\beta$ stack: | × |

(10) *The 10th token is + — the operator + (addition).* The operator gets assigned the precedence value of 2. The operator is not immediately pushed into the $\beta$ stack because the × operator from the 8th step has a higher precedence (5) than + (2). The operator with the higher precedence at the top of the stack must go and get registered; only then can + takes its place.

| | |
|---|---|
| $\alpha$ stack: | 21 |
| $\beta$ stack: | + |

(11) *The 11th token is* `2` *— the operand 2.* The operand is re gistered into the postfix processor, and into the $\alpha$ stack, awaiting an operator to evaluate it. In truth, there is a + operator and it is already registered by the calculator. However, the infix processor is still holding it up in the $\beta$ stack because it assumes the now-false assumption that there are to be more tokens. The next step removes this false assumption.

| | |
|---|---|
| $\alpha$ stack: | 21 2 |
| $\beta$ stack: | + |

(12) *The* `.INEVAL` *request is called.* When such a request is called, all operators from the $\beta$ stack are cleared and registered one-by-one into the postfix processor at the order which they are popped. Without any error checking, it does this until the $\beta$ stack is empty. Of course, this requires a loop, which is implemented by primitively recursive macro calls in Troff. When all the operators are registered into the postfix processor and the $\beta$ stack is emptied, the final step of this step is to print the supposedly-final value in the $\alpha$ stack, which is left after all the evaluation done by the postfix processor.

$\alpha$ stack: 23
$\beta$ stack:

This is a very specific run of the shunting-yard algorithm and is provided only as an example. For a more general and detailed account of the algorithm, see the Wikipedia article about it.[10]

It is not advised to register any more tokens into the infix processor after `.INEVAL` has been called.

You may have also noticed that the infix processor does outsource some logic to the postfix processor and the stack implementations to handle. This is a simple example of the UNIX philosophy:

> Write small, simple programs that each do one thing and do it well and put them together to perform the desired computation, instead of writing one big program that combines all the logic into one monolithic package. In this way, the logic of the entire ordered set of programs can be looked at and maintained separately, and the logic can also be rearranged and put together differently to achieve a different result if so desired later on down the line.

### 4.3.3. Postfix notation

Expressions set in postfix are processed using the reverse polish notation (postfix) processor in `rpn.roff`. The algorithm implemented is a very trivial reverse polish notation parser that one can get from the Wikipedia page about it.[11] Although it is the heart and soul of the entire calculator, it is extremely simple. So simple in fact, that `rpn.roff` is usually approximately half the size of `infix.roff`. Thus, I do not expect that this subsubsection should be as lengthy as the one before.

The infix processor requires two stacks for it to really function as it should. By contrast, the postfix processor requires only one stack. This stack, namely the $\alpha$ stack, could be thought of as a workspace which can be used by the postfix processor to store intermediate values to be evaluated or/and the (intermediate) result itself; it never stores operators, as operators merely serve to instruct the processor to evaluate expressions in the stack. Because of its function, this stack is often called the “evaluation stack” equivalently. Yes, this is the same stack used by `infix.roff`, but this stack really only stores operand tokens!

Since Troff lacks the means really read and extract tokens from an input string like JavaScript can, we must do the work of extracting all the tokens in our expressions by ourselves. For each token we extract, we must register it with the `.RPNPUSH` *`t`* request, dubbed from *Reverse Polish Notation Push*, and keep doing so until the end of your expression. Do make sure that your input expression is correct as you “push” them, so that you get the results you want without any unpleasant surprises.

The processor also has absolutely no way of knowing if you have finished pushing your tokens. Once you know you have finished, you must explicitly ask for a result with the `.RPNPRINT` request. Notice that it is dubbed from *Reverse Polish Notation* ***Print***. Unlike the infix processor, evaluation can be done immediately whenever possible, but you just need to ask the calculator to print it explicitly.

Consider the same input expression from the last section: $(4 + 5 - 2) \times 3 + 2$. This expression could be converted by hand to postfix like so: $4\ 5\ 2\ -\ +\ 3 \times 2\ +$.[12] Assuming the converted expression has been broken down into tokens and the tokens are prepared in an input file to the processor like the one in § 4.2. Input§2. Postfix notation, the following pseudo-description shows how the algorithm runs.

(1) *1st token: 4 — the operand 4.* This gets pushed into the stack immediately . In fact, ***all*** operands get pushed into the stack immediately!

$\alpha$ stack: 4

[10] The delayed text should provide this. Refer to the Bibliography for the link.

[11] The delayed text should provide this too. Refer to the Bibliography for the link.

[12] Equivalently, it could also be $4\ 5 + 2 - 3 \times 2\ +$. However, we do not assume this ordering in the pseudo-description. Remember that order of the tokens is very important when explaining the algorithm!

(2) *2nd token: 5 — the operand 5.* It is pushed into the stack.

$\alpha$ stack: 4 5

(3) *3rd token: 2 — the operand 2.* It is pushed into the stack.

$\alpha$ stack: 4 5 2

(4) *4th token: − — the subtraction operator.* The last two items of the stacks are popped and stored in two registers, `O2` and `O1` procedurally in the order at which they are popped. The numbers in the two registers are subtracted with each other as `O1 - O2`. The result is pushed back into the $\alpha$ stack. Since there are more tokens to be read, this is just an intermediate result, awaiting to be evaluated again by some other operator.

$\alpha$ stack: 4 3

(5) *5th token: + — the addition operator.* The last two items of the stacks are popped and stored in two registers, `O2` and `O1` procedurally in the order at which they are popped. The numbers in the two registers are added to each other as `O1 + O2`. The result is pushed back into the $\alpha$ stack. Since there are more tokens to be read, this is just an intermediate result, awaiting to be evaluated again by some other operator. *By now, you should have already known all the quirks and caveats of this algorithm. Subsequent steps will not repeat words.*

$\alpha$ stack: 7

(6) *6th token: 3 — the operand 3.*

$\alpha$ stack: 7 3

(7) *7th token: × — the multiplication operator.*

$\alpha$ stack: 21

(8) *8th token: 2 — the operand 2.*

$\alpha$ stack: 21 2

(9) *9th token: + — the addition operator.*

$\alpha$ stack: 23

As you can see, there are less steps taken by the postfix processor doing the actual evaluation than there are steps taken by the infix processor converting expressions into an ordered set of postfix tokens! This generally means that the evaluation in postfix is faster than in infix, although their time complexity is about the same at O(n) for *n* tokens. The results we arrive at are the same nonetheless.

See the Wikipedia article on *Reverse Polish Notation* for a more general and detailed explanation of the algorithm.

### 4.3.4. Practical usage

None.

Just kidding.

You may use a TROFF compiler like GROFF to compile your input file. You should not need to use an typesetting macros (-me, -ms, or -mm), as they are only good for typesetting documents. In this section, you are assumed to use GROFF.

To compile your input file (assuming it is saved to `input.roff`) and output the evaluation result as a Postscript file (named `output.ps`) that can be viewed by Ghostscript or printed[13], run:

```
groff -Tps input.roff > output.ps
```

If you would like a PDF output instead, replace `-Tps` with `-Tpdf`. That doesn't work on some systems, though, namely Cygwin. If you do want it in Postscript, you need not explicitly specify the `-Tps` option.

To compile your input file with the same name and output the result to the terminal for convenience, run:

```
groff -Tutf8 input.roff | head -n 5
```

This will output directly to stdout. Without the `head` utility, the output could exceed your terminal's current buffer, for nroff leaves paper-sized blank spaces at currently inconvenient places.

## 4.4. Output

Assuming the processing went well (your expressions are syntactically and logically correct and no errors/warnings are output to stderr), you should get the same results no matter if the same expression is set in infix or postfix.

For example, if we use the same input expression as before, the output to Postscript or the terminal should be the result of that expression, which is:

```
23
```

on the terminal. It should always be 23, regardless of whether you parsed as infix or postfix to their respective processors. If you think about it, it should be like that. The postfix processor does the actual evaluation and the infix processor's *only* job is to find the equivalent postfix expression for the infix expression. This thus leads to the conclusion that:

$$(4 + 5 - 2) \times 3 + 2 = 4\ 5\ 2 - + \ 3 \times 2 + = 23$$

## 5. Motivation

The motivation behind making a Troff calculator was initially more practical than esoteric. There were actual practical uses for it, but it seemed to prove more esoteric later down the line.

As a 17-year-old high school student, there are many things to think about: my exams and my academic future in university, to name a few. As I kept thinking about those things for a while, I started getting sick, had nightmares, anxiety, and some weird depression. To kind of shut the voices inside my head up, I tried studying as much as I can on many things, really, just to prove that I am better than what my mind has led me on to believe; not only that, it also feels a little good being better than other people in my class.[14]

As my dream since childhood has always been to study computer science, be a good computer scientist and programmer, and work in the relevant faculties in university; I really have to push it. I have been trying to be at the bleeding-edge of all my curricula. Part of my efforts eminates through as my extensive use of the UNIX-like environments (Gentoo, OpenBSD, and similars), and programming weird stuff and

[13] Think of the trees, bastard. By the way, if you are reading this README from physical paper, you also qualify for a bastard too!

[14] Hahahahaaa... Uh... It was just a joke. Jk jk, k? hehe.

going into theory.

One of the things in computer science that got me really interested was the Reverse Polish Notation. It was amazing when I heard of it on Computerphile, and so I looked it up on Wikipedia. See, it really got me on edge, but I found the lack of calculators and tools rather really discouraging. I don't want to see myself literally programming in stack-based languages like Postscript or FORTH, and I do not want to download weird calculator apps on my phone.

So, it was an "all in or nothing" situation, as they say. As there were no programs that could satisfy me, I decided to program this calculator by myself, using the only "programming language" that I actively used at the time: Troff.

*You know, I really thought at the time that I was out of my mind!* "T roff?! Are you serious?" I exclaimed to myself, "Troff is a typesetting language used to prepare documents, ***not*** an actual programming language!" I sat for a while and agreed to disagree, "Well, if it is hell, it could teach me something about computing." "Oh really? How?" "I will have a much deeper, less abstract understanding of computation," speaking like the smartest girl in the world, *duh*!

And so I spent my holiday programming this thing, I guess. At the time, the infix processor wasn't there yet, so we will only talk about the postfix processor first. It worked way better than I expected! So, I tried giving it some test input expressions in postfix. It worked spectacularly well, really. I made more mistakes checking by hand (2–3 times) than it did (0 times)! I do have to say that I am bad at mathematics, though.

After a few hours of finally experimenting with the Reverse Polish notation by using my program as intended, its non-esoteric use case has gone past and I got the hang of it. So, I decided to upload the entire original code base as *katt64/rpn-calculator* on Github. I felt I have already gotten used to Reverse Polish just by playing with it for several hours, and so I decided to do one better: "let's make an infix parser in Troff!" Initially, I thought that this was too ambitious a goal, so I really just settled on making a parser that converted infix expressions into postfix for the already-written RPN processor instead.

All of that got me into the shunting-yard algorithm for a while. Then, after experimenting with the algorithm by hand, I implemented it using stacks and all that right within Troff, and linked it to the RPN processor I had already written. To finish it off, I wrote the input source file of some expressions set in infix notation. Of course, I was anxious that it would not work; I crossed my fingers, eyes, and legs, and just hoped for the best as the Troff source file was compiling and the Postscript file was loading. In disbelief, it totally worked! The results for all my test cases were as expected from the program. However, literally for days, I really couldn't look at the Troff macros I had written; it was utterly complicated, with registers and what-the-hells flying every fucking place imaginable and unimaginable, and it was worse than a write-only program.

Admittedly, the shunting-yard algorithm implementation was purely estoeric, to demonstrate that Troff could very well be Turing-complete. Since I knew that this calculator had grown beyond its RPN-usage into an esoteric world, I deicded to upload an improved version of the entire code base into this new Github repository: *katt64/troff-calculator*. "It's not just an RPN-calculator, it is a real calculator now!" I said.

By the way, I did send an email to Professor Brain W. Kernighan about this calculator not so long ago. I explained what I had done, how I got it to work, how it worked, the motivation behind it, and I gave him the link to the code repository on Github, within short email of less than 10 paragraphs. In the email, I also attached a few pictures of it working and a video demonstrating its usage. Within a few days' time, he gave me a prompt reply saying, "It was fun looking at the code," and he also said that "***T roff is indeed Turing-complete.***" Oh well, that got me off the chair, flying to the ceiling, literally! :p Of course, I have the exact email correspondence burried in my inbox somewhere, but even if I find it, I will not show it to anyone.

To this day, I still think that I was totally crazily out of my mind for programming a calculator in Troff. Though, I did learn quite a few new things in computer science, and I would love to do the exact same things again.

Having implemented stacks that could be used by many algorithms in Troff, I do agree with Professor Brian W. Kernighan that Troff is indeed Turing-complete. This also means that it could be possible to implement the Cellular Automaton Rule 110, which is said to be Turing-complete.[6]

## 6. Bugs

Not all programs are perfect: this one is not perfect either. There are notably a few bugs that I have seen in the program but I never really have the time or effort to get back and fix it, because I just don't feel like it matters. Below lists a bunch of bugs that I have seen so far. Of course, there could be many other bugs out there that I have not seen, only waiting to be unpleasantly discovered. If you want me to fix these bugs, you might as well send me some love and support so I can feel a little more encouraged to fix them.

(1) In the implementation of the shunting-yard algorithm in `infix.roff`, when an operator is registered and its precedence is compared, if the operator at the top of the $\beta$ stack is of greater or equal precedence to the newly-registered operator, then the existing operator is popped from stack, but only once. That is, if there (is|are) still operator(s?) in the $\beta$ stack of greater or equal precedence to the newly-registered operator even after the popping has been done, no more operators will be popped and that operator is pushed regardless. According to the definition of the algorithm, operators should be popped until the stack counter is 0 or the last operator in the stack is of lower precedence than the newly-registered operator (requires a loop).

If you have found any more bugs, please report them as Issues on the repository's Github site. From there, the issues will be looked at my maintainers and discussed. Please follow adequate etiquette and give as detailed a bug report as possibly can, repeating words and sentences a million times if necessary; this is so that we can get as detailed a picture of the problem as possible.

## 7. Todo

I will try to make the user interface a little bit better. At present, tokens need to be registered one request at a time in the input file. I may create a new request that does all this job; it should scan through its argument list and internally register them. Then, one need no longer parse tokens line-by-line, but rather word-by-word on the same line, which makes for a more convenient and readable input file. Here is an example of a future user-accessible request:

```
.RPNPARSE 9 3 / 5 2 + *
```

This will immediately evaluate and print the result of the RPN expression $9\ 3 \div 5\ 2 + \times$ which must exist on the same line, with tokens separated by spaces. No existing requests will be removed, as `.RPNPARSE` essentially acts as a user-friendly wrapper and ergo extensively depends on many of those requests. A similar request for the infix processor will also be made, called `.INPARSE`.

I will also try to set this README document into reStructuredText by hand if I have time. As far as I know, no utility has the ability to convert documents set in the *-me* macros into simpler document formats like Markdown or reStructuredText, not even Pandoc.

## 8. Author

Stephanie Björk (Katt) <katt16777216@gmail.com>

You are welcome to ask any questions, give constructive feedback, or just say "Hi" and talk about how much you love/hate this esoteric project. :p I usually check my inbox every once in a few days, so make your emails count, and please be patient. Please do ***not*** send hurtful comments that could do more harm than good to me. I've been through quite enough of those things and I definitely do not want to have an argument I can't ever seem to win with you or any academic-big-guys.

If it is an emergency and you need to get in touch with me immediately, please add me on Snapchat at `suttiwit`. Unlike email, it could take a mere matter of minutes for me to reply. ***Do not*** send nudes or selfies, unless you want to be blocked permanently.

## 9. License

This calculator is licensed in the *Do What The Fuck You Want To Public License* version 2. The entire license is as follows (also see `LICENSE` in the repository):

So, you just do what the fuck you want to, okay? If you have a problem with that, it is yours. Have a nice day.

## Bibliography

[1] *troff-calculator* on Github: https://github.com/katt64/troff-calculator

[2] *Reverse Polish notation* on Wikipedia: https://en.wikipedia.org/wiki/Reverse_Polish_notation

[3] *Stephanie Björk* (Personal website): https://katt64.github.io

[4] *rpn-calculator* on Github: https://github.com/katt64/rpn-calculator

[5] *Shunting-yard algorithm* on Wikipedia: https://en.wikipedia.org/wiki/Shunting-yard_algorithm

[6] *Rule 110 § The proof of universality* on Wikipedia: https://en.wikipedia.org/wiki/Rule_110#The_proof_of_universality

This README file was typeset in TROFF using the *-me* macros to set the content of the document and the *eqn* preprocessor to set mathematics, on November 27, 2017.

*Dedicated to...*

*My mom*
*“You never mind me staying up late, writing this documentation till 4 AM the next day during my holidays and weekends.”*

*Bjørk*
*“You’ve saved my life.”*

## Abstract

This is the official README file for the entire project, *katt64/troff-calculator*. It aims to document every corner piece and subtleties that exist within the project. However, no documentation is as good as looking at the code itself, so it is advisable that the code be looked at in order to really get the idea of the program's operation.

In this README, an accurate description of the project; the program's capabilities and limitations; a precise description of the program's operation; some notes on the motivation behind such a project; any problems or bugs found and how they can be reported; and some notes about the author and licensing will be found, as well as many other important things.

For basic, non-technical usage, only §§ Operation§Prerequisites, Operation§Input, Operation§Processing§Practical usage, and Operation§Output need be read, and other more technical material can be omitted. Refer to the Table of contents for the page numbers thither.

# Table of contents