CS320 Fall2023 Project Part 1

November 2023

1   Overview

Stack-oriented programming languages utilize one or more stack data structures which the programmer manipulates to perform computations. The specification below lays out the syntax and semantics of such a language which you will need to implement an evaluator for, taking a program and evaluating it into an OCaml value. You must implement a function:

```
interp : string -> string list option
```

Where the input string is some (possibly ill-formed) stack-language program and the result is a list of things "printed" by the program during its evaluation.

2 Grammar

For part 1, you will need to support the following grammar. The grammar specifies the *form* of a valid program. Any *invalid* program, one which is not derivable from the grammar, cannot be given meaning and evaluted. In the case of an invalid program, `interp` should return `None`. $\langle prog \rangle$ is the starting symbol.

2.1 Constants

$\langle digit \rangle$ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\langle nat \rangle$ ::= $\langle digit \rangle$ | $\langle digit \rangle \langle nat \rangle$

$\langle int \rangle$ ::= $\langle nat \rangle$ | -$\langle nat \rangle$

$\langle bool \rangle$ ::= True | False

$\langle const \rangle$ ::= $\langle int \rangle$ | $\langle bool \rangle$ | Unit

2.2 Programs

$\langle prog \rangle$ ::= $\langle coms \rangle$

$\langle com \rangle$ ::= Push $\langle const \rangle$ | Pop | Trace
        | Add | Sub | Mul | Div
        | And | Or | Not
        | Lt | Gt

$\langle coms \rangle$ ::= ϵ | $\langle com \rangle$; $\langle coms \rangle$

Note: ϵ is the empty symbol. We use ϵ to refer to empty strings or empty lists depending on context.

3 Operational Semantics

For part 1, you will need to support the following operational semantics. The operational semantics specifies the *meaning* of a valid program. For the stack-language, a program is evaluated using a stack and it produces a trace. Once we have fully evaluated the program, we return the resulting trace from `interp`.

3.1 Configuration of Programs

A program configuration is of the following form.

$$[\, S \mid T \,]\ P$$

- S: (**S**tack) stack of intermediate values

- T: (**T**race) list of strings logging the program trace

- P: (**P**rogram) program commands to be interpreted

Examples:

$$[\, \epsilon \mid \epsilon \,]\ \texttt{Push True}; \texttt{Not}; \texttt{Push 1}; \texttt{Lt}; \epsilon \tag{1}$$

$$[\, 1 :: 2 :: \epsilon \mid \texttt{"True"} :: \texttt{"0"} :: \epsilon \,]\ \texttt{Push True}; \texttt{Push 9}; \texttt{Pop}; \epsilon \tag{2}$$

$$[\, 0 :: \texttt{True} :: \epsilon \mid \epsilon \,]\ \texttt{Push 10}; \texttt{Push 9}; \texttt{Add}; \texttt{Trace}; \epsilon \tag{3}$$

$$[\, \texttt{True} :: \texttt{False} :: 321 :: \epsilon \mid \texttt{"123"} :: \texttt{"False"} :: \epsilon \,]\ \texttt{Pop}; \texttt{Pop}; \texttt{Trace}; \texttt{Gt}; \epsilon \tag{4}$$

3.2 Program Reduction

The operational semantics of the language is defined in terms of the following single step relation.

$$[\, S_1 \mid T_1 \,]\ P_1 \rightsquigarrow [\, S_2 \mid T_2 \,]\ P_2$$

In one step, program configuration $[\, S_1 \mid T_1 \,]\ P_1$ evaluates to $[\, S_2 \mid T_2 \,]\ P_2$. For configurations where $P = \epsilon$, we say that evaluation has terminated as there is no program left to interpret. In this case, return trace T as the final result of your `interp` function.

3.3 Push

Given any constant c, the command `Push` c pushes c onto the current stack S. `Push` never fails.

$$\frac{}{[\, S \mid T \,]\ \texttt{Push } c; P \rightsquigarrow [\, c :: S \mid T \,]\ P} \text{ Push}$$

Examples:

- $[\, 1 :: \texttt{True} :: \epsilon \mid \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\ \texttt{Push 2}; \epsilon \rightsquigarrow [\, 2 :: 1 :: \texttt{True} :: \epsilon \mid \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 1 :: \texttt{True} :: \epsilon \mid \texttt{"5"} :: \epsilon \,]\ \texttt{Push True}; \epsilon \rightsquigarrow [\, \texttt{True} :: 1 :: \texttt{True} :: \epsilon \mid \texttt{"5"} :: \epsilon \,]\ \epsilon$

3.4 Pop

Given a stack of the form $c :: S$ (constant c is on top of S), the `Pop` command removes c and leaves the rest of stack S unmodified.

The `Pop` command has 1 fail state.

1. POPERROR: The stack is empty ($S = \epsilon$).

When `Pop` fails, the string `"Panic"` is prepended to the trace and the program terminates.

$$\frac{}{[\, c :: S \mid T \,]\; \texttt{Pop}; P \rightsquigarrow [\, S \mid T \,]\; P} \text{ POPSTACK} \qquad \frac{}{[\, \epsilon \mid T \,]\; \texttt{Pop}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,]\; \epsilon} \text{ POPERROR}$$

Examples:

- $[\, 1 :: \texttt{True} :: \epsilon \mid \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\; \texttt{Pop}; \epsilon \rightsquigarrow [\, \texttt{True} :: \epsilon \mid \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\; \epsilon$

- $[\, \epsilon \mid \texttt{"5"} :: \epsilon \,]\; \texttt{Pop}; \texttt{Push } 12; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"5"} :: \epsilon \,]\; \epsilon$

3.5 Trace

Given a stack of the form $c :: S$ where c is any valid constant, the `Trace` command removes c from the stack and puts a `Unit` constant onto the stack. The string representation of c as determined by the *toString* function to prepended to the trace.

The `Trace` command has 1 fail state.

1. TRACEERROR: The stack is empty ($S = \epsilon$).

When `Trace` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

$$\frac{}{[\, c :: S \mid T \,]\; \texttt{Trace}; P \rightsquigarrow [\, \texttt{Unit} :: S \mid \mathit{toString}(c) :: T \,]\; P} \text{ TRACESTACK} \qquad \frac{}{[\, \epsilon \mid T \,]\; \texttt{Trace}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,]\; \epsilon} \text{ TRACEERROR}$$

The *toString* function is a special function which you must define to convert constant values into their string representations. The following equations illustrate the strings expected for typical inputs.

$$\mathit{toString}(123) = \texttt{"123"} \tag{1}$$
$$\mathit{toString}(\texttt{True}) = \texttt{"True"} \tag{2}$$
$$\mathit{toString}(\texttt{False}) = \texttt{"False"} \tag{3}$$
$$\mathit{toString}(\texttt{Unit}) = \texttt{"Unit"} \tag{4}$$

Examples:

- $[\, 1 :: \texttt{True} :: \epsilon \mid \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\; \texttt{Trace}; \epsilon \rightsquigarrow [\, \texttt{Unit} :: \texttt{True} :: \epsilon \mid \texttt{"1"} :: \texttt{"Unit"} :: \texttt{"False"} :: \epsilon \,]\; \epsilon$

- $[\, \epsilon \mid \texttt{"5"} :: \epsilon \,]\; \texttt{Trace}; \texttt{Push } 12; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"5"} :: \epsilon \,]\; \epsilon$

4

3.6 Add

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the Add command removes i and j from the stack and puts their sum $(i + j)$ onto the stack.

 The Add command has 3 fail states.

1. ADDERROR1: Either i or j is not an integer.

2. ADDERROR2: The stack is empty $(S = \epsilon)$.

3. ADDERROR3: The stack has only 1 element $(S = c :: \epsilon)$.

When Add fails, the stack is cleared, the string "Panic" is prepended to the trace and the program terminates.

ADDSTACK
$$\frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \,]\ \mathtt{Add}; P \rightsquigarrow [\, (i + j) :: S \mid T \,]\ P}$$

ADDERROR1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \,]\ \mathtt{Add}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

ADDERROR2
$$\frac{}{[\, \epsilon \mid T \,]\ \mathtt{Add}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

ADDERROR3
$$\frac{}{[\, c :: \epsilon \mid T \,]\ \mathtt{Add}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

Examples:

- $[\, 4 :: 5 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Add}; \epsilon \rightsquigarrow [\, 9 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Add}; \mathtt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Add}; \mathtt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon \,]\ \epsilon$

3.7 Sub

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the Sub command removes i and j from the stack and puts their difference $(i - j)$ onto the stack.

 The Sub command has 3 fail states.

1. SUBERROR1: Either i or j is not an integer.

2. SUBERROR2: The stack is empty $(S = \epsilon)$.

3. SUBERROR3: The stack has only 1 element $(S = c :: \epsilon)$.

When Sub fails, the stack is cleared, the string "Panic" is prepended to the trace and the program terminates.

SUBSTACK
$$\frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \,]\ \mathtt{Sub}; P \rightsquigarrow [\, (i - j) :: S \mid T \,]\ P}$$

SUBERROR1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \,]\ \mathtt{Sub}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

SUBERROR2
$$\frac{}{[\, \epsilon \mid T \,]\ \mathtt{Sub}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

SUBERROR3
$$\frac{}{[\, c :: \epsilon \mid T \,]\ \mathtt{Sub}; P \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: T \,]\ \epsilon}$$

Examples:

- $[\, 4 :: 5 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Sub}; \epsilon \rightsquigarrow [\, -1 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Sub}; \mathtt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \epsilon \mid \text{"False"} :: \epsilon \,]\ \mathtt{Sub}; \mathtt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon \,]\ \epsilon$

3.8   Mul

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the `Mul` command removes i and j from the stack and puts their product $(i \times j)$ onto the stack.

The `Mul` command has 3 fail states.

- MULERROR1: Either i or j is not an integer.

- MULERROR2: The stack is empty $(S = \epsilon)$.

- MULERROR3: The stack has only 1 element $(S = c :: \epsilon)$.

When `Mul` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

MULSTACK
$$\frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \,]\ \texttt{Mul}; P \rightsquigarrow [\, (i \times j) :: S \mid T \,]\ P}$$

MULERROR1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,]\ \epsilon}$$

MULERROR2
$$\frac{}{[\, \epsilon \mid T \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,]\ \epsilon}$$

MULERROR3
$$\frac{}{[\, c :: \epsilon \mid T \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,]\ \epsilon}$$

Examples:

- $[\, 4 :: 5 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,]\ \texttt{Mul}; \epsilon \rightsquigarrow [\, 20 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,]\ \texttt{Mul}; \texttt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon \,]\ \epsilon$

- $[\, 4 :: \epsilon \mid \texttt{"False"} :: \epsilon \,]\ \texttt{Mul}; \texttt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon \,]\ \epsilon$

3.9 Div

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the `Div` command removes i and j from the stack and puts their quotient $(i \div j)$ onto the stack.

The `Div` command has 4 fail states.

1. DivError0: Both i and j are integers and $j = 0$.

2. DivError1: Either i or j is not an integer.

3. DivError2: The stack is empty $(S = \epsilon)$.

4. DivError3: The stack has only 1 element $(S = c :: \epsilon)$.

When `Div` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

DivStack
$$\frac{i \text{ and } j \text{ are both integers, } j \neq 0}{[\, i :: j :: S \mid T \,] \, \texttt{Div}; P \rightsquigarrow [\, (i \div j) :: S \mid T \,] \, P}$$

DivError0
$$\frac{i \text{ is an integer}}{[\, i :: 0 :: S \mid T \,] \, \texttt{Div}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,] \, \epsilon}$$

DivError1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \,] \, \texttt{Div}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,] \, \epsilon}$$

DivError2
$$\frac{}{[\, \epsilon \mid T \,] \, \texttt{Div}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,] \, \epsilon}$$

DivError3
$$\frac{}{[\, c :: \epsilon \mid T \,] \, \texttt{Div}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \,] \, \epsilon}$$

Examples:

- $[\, 16 :: 8 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,] \, \texttt{Div}; \epsilon \rightsquigarrow [\, 2 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,] \, \epsilon$

- $[\, 16 :: 0 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,] \, \texttt{Div}; \texttt{Push Unit}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon \,] \, \epsilon$

- $[\, 16 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon \,] \, \texttt{Div}; \texttt{Add}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon \,] \, \epsilon$

- $[\, 4 :: \epsilon \mid \texttt{"False"} :: \epsilon \,] \, \texttt{Div}; \texttt{Trace}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon \,] \, \epsilon$

3.10   And

Given a stack of the form $a :: b :: S$ where both a and b are boolean values, the `And` command removes a and b from the stack and puts their conjunction $(a \wedge b)$ onto the stack.

The `And` command has 3 fail states.

1. ANDERROR1: Either a or b is not a boolean.

2. ANDERROR2: The stack is empty $(S = \epsilon)$.

3. ANDERROR3: The stack has only 1 element $(S = c :: \epsilon)$.

When `And` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

$$\frac{\text{ANDSTACK} \qquad a \text{ and } b \text{ are both booleans}}{[\,a :: b :: S \mid T\,]\ \texttt{And}; P \rightsquigarrow [\,(a \wedge b) :: S \mid T\,]\ P} \qquad \frac{\text{ANDERROR1} \qquad a \text{ or } b \text{ is not a boolean}}{[\,a :: b :: S \mid T\,]\ \texttt{And}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

$$\frac{\text{ANDERROR2}}{[\,\epsilon \mid T\,]\ \texttt{And}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon} \qquad \frac{\text{ANDERROR3}}{[\,c :: \epsilon \mid T\,]\ \texttt{And}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

Examples:

- $[\,\texttt{True} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{And}; \epsilon \rightsquigarrow [\,\texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \epsilon$

- $[\,\texttt{False} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{And}; \texttt{Trace}; \epsilon \rightsquigarrow [\,\texttt{False} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{Trace}; \epsilon$

- $[\,\texttt{True} :: 4 :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{And}; \texttt{Pop}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,]\ \epsilon$

3.11   Or

Given a stack of the form $a :: b :: S$ where both a and b are boolean values, the `Or` command removes a and b from the stack and puts their disjunction $(a \vee b)$ onto the stack.

The `Or` command has 3 fail states.

1. ORERROR1: Either a or b is not a boolean.

2. ORERROR2: The stack is empty $(S = \epsilon)$.

3. ORERROR3: The stack has only 1 element $(S = c :: \epsilon)$.

When `Or` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

$$\frac{\text{ORSTACK} \qquad a \text{ and } b \text{ are both booleans}}{[\,a :: b :: S \mid T\,]\ \texttt{Or}; P \rightsquigarrow [\,(a \vee b) :: S \mid T\,]\ P} \qquad \frac{\text{ORERROR1} \qquad a \text{ or } b \text{ is not a boolean}}{[\,a :: b :: S \mid T\,]\ \texttt{Or}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

$$\frac{\text{ORERROR2}}{[\,\epsilon \mid T\,]\ \texttt{Or}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon} \qquad \frac{\text{ORERROR3}}{[\,c :: \epsilon \mid T\,]\ \texttt{Or}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

Examples:

- $[\,\texttt{True} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{Or}; \epsilon \rightsquigarrow [\,\texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \epsilon$

- $[\,\texttt{False} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{Or}; \texttt{Trace}; \epsilon \rightsquigarrow [\,\texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{Trace}; \epsilon$

- $[\,\texttt{True} :: 4 :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \texttt{Or}; \texttt{Pop}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,]\ \epsilon$

3.12   Not

Given a stack of the form $a :: S$ where a is a boolean values, the `Not` command removes a from the stack and puts its negation ($\neg a$) onto the stack.

The `Not` command has 2 fail states.

1. NOTERROR1: a is not a boolean.

2. NOTERROR2: The stack is empty ($S = \epsilon$).

When `Not` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

NOTSTACK
$$\frac{a \text{ is a boolean}}{[\,a :: S \mid T\,] \; \texttt{Not}; P \rightsquigarrow [\,(\neg a) :: S \mid T\,] \; P}$$

NOTERROR1
$$\frac{a \text{ is not a boolean}}{[\,a :: S \mid T\,] \; \texttt{Not}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,] \; \epsilon}$$

NOTERROR2
$$\frac{}{[\,\epsilon \mid T\,] \; \texttt{Not}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,] \; \epsilon}$$

Examples:

- $[\,\texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Not}; \epsilon \rightsquigarrow [\,\texttt{False} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \epsilon$

- $[\,4 :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Not}; \texttt{Pop}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,] \; \epsilon$

- $[\,\epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Not}; \texttt{Add}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,] \; \epsilon$

3.13   Lt

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the `Lt` command removes i and j from the stack and puts the **boolean** result of their comparison ($i < j$) onto the stack.

The `Lt` command has 3 fail states.

1. LTERROR1: Either i or j is not an integer.

2. LTERROR2: The stack is empty ($S = \epsilon$).

3. LTERROR3: The stack has only 1 element ($S = c :: \epsilon$).

When `Lt` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

LTSTACK
$$\frac{i \text{ and } j \text{ are both integers}}{[\,i :: j :: S \mid T\,] \; \texttt{Lt}; P \rightsquigarrow [\,(i < j) :: S \mid T\,] \; P}$$

LTERROR1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\,i :: j :: S \mid T\,] \; \texttt{Lt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,] \; \epsilon}$$

LTERROR2
$$\frac{}{[\,\epsilon \mid T\,] \; \texttt{Lt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,] \; \epsilon}$$

LTERROR3
$$\frac{}{[\,c :: \epsilon \mid T\,] \; \texttt{Lt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,] \; \epsilon}$$

Examples:

- $[\,4 :: 5 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Lt}; \epsilon \rightsquigarrow [\,\texttt{True} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \epsilon$

- $[\,5 :: 5 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Lt}; \epsilon \rightsquigarrow [\,\texttt{False} :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \epsilon$

- $[\,4 :: \texttt{True} :: \texttt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,] \; \texttt{Lt}; \texttt{Trace}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,] \; \epsilon$

3.14   Gt

Given a stack of the form $i :: j :: S$ where both i and j are integer values, the `Gt` command removes i and j from the stack and puts the **boolean** result of their comparison $(i > j)$ onto the stack.

The `Gt` command has 3 fail states.

1. GtError1: Either i or j is not an integer.

2. GtError2: The stack is empty $(S = \epsilon)$.

3. GtError3: The stack has only 1 element $(S = c :: \epsilon)$.

When `Gt` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

GtStack
$$\frac{i \text{ and } j \text{ are both integers}}{[\,i :: j :: S \mid T\,]\ \mathtt{Gt}; P \rightsquigarrow [\,(i > j) :: S \mid T\,]\ P}$$

GtError1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\,i :: j :: S \mid T\,]\ \mathtt{Gt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

GtError2
$$\frac{}{[\,\epsilon \mid T\,]\ \mathtt{Gt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

GtError3
$$\frac{}{[\,c :: \epsilon \mid T\,]\ \mathtt{Gt}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T\,]\ \epsilon}$$

Examples:

- $[\,4 :: 5 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \mathtt{Gt}; \epsilon \rightsquigarrow [\,\mathtt{False} :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \epsilon$

- $[\,10 :: 5 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \mathtt{Gt}; \epsilon \rightsquigarrow [\,\mathtt{True} :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \epsilon$

- $[\,5 :: 5 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \mathtt{Gt}; \epsilon \rightsquigarrow [\,\mathtt{False} :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \epsilon$

- $[\,4 :: \mathtt{True} :: \mathtt{Unit} :: \epsilon \mid \texttt{"False"} :: \epsilon\,]\ \mathtt{Gt}; \mathtt{Trace}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"False"} :: \epsilon\,]\ \epsilon$

4 Full Examples

- Compute the polynomial $x^2 - 4x + 7$ at 3:

```
Push 3;
Push 3;
Mul;
Push -4;
Push 3;
Mul;
Add;
Push 7;
Add;
Trace;
```

  Result: Some ["4"]

- De Morgan's Law:

```
Push False;
Push False;
And;
Not;
Trace;
Push False;
Not;
Push False;
Not;
Or;
Trace;
```

  Result: Some ["True"; "True"]

- x^2 is monotonic:

```
Push 2;
Push 2;
Mul;
Push 3;
Push 3;
Mul;
Gt;
Trace;
```

  Result: Some ["True"]