

# 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 evaluated. In the case of an invalid program, **interp** should return **None**.  $\langle prog \rangle$  is the starting symbol.

### 2.1 Constants

```
 $\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$   
 $\langle nat \rangle ::= \langle digit \rangle \mid \langle digit \rangle \langle nat \rangle$   
 $\langle int \rangle ::= \langle nat \rangle \mid -\langle nat \rangle$   
 $\langle bool \rangle ::= \text{True} \mid \text{False}$   
 $\langle const \rangle ::= \langle int \rangle \mid \langle bool \rangle \mid \text{Unit}$ 
```

### 2.2 Programs

```
 $\langle prog \rangle ::= \langle coms \rangle$   
 $\langle com \rangle ::= \text{Push } \langle const \rangle \mid \text{Pop} \mid \text{Trace}$   
            $\mid \text{Add} \mid \text{Sub} \mid \text{Mul} \mid \text{Div}$   
            $\mid \text{And} \mid \text{Or} \mid \text{Not}$   
            $\mid \text{Lt} \mid \text{Gt}$   
 $\langle coms \rangle ::= \epsilon \mid \langle com \rangle; \langle coms \rangle$ 
```

Note:  $\epsilon$  is the empty symbol. We use  $\epsilon$  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:

$$[\epsilon \mid \epsilon] \text{Push True; Not; Push 1; Lt; } \epsilon \quad (1)$$

$$[1 :: 2 :: \epsilon \mid \text{"True"} :: \text{"0"} :: \epsilon] \text{Push True; Push 9; Pop; } \epsilon \quad (2)$$

$$[0 :: \text{True} :: \epsilon \mid \epsilon] \text{Push 10; Push 9; Add; Trace; } \epsilon \quad (3)$$

$$[\text{True} :: \text{False} :: 321 :: \epsilon \mid \text{"123"} :: \text{"False"} :: \epsilon] \text{Pop; Pop; Trace; Gt; } \epsilon \quad (4)$$

#### 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.

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

Examples:

- $[1 :: \text{True} :: \epsilon \mid \text{"Unit"} :: \text{"False"} :: \epsilon] \text{Push 2; } \epsilon \rightsquigarrow [2 :: 1 :: \text{True} :: \epsilon \mid \text{"Unit"} :: \text{"False"} :: \epsilon] \epsilon$
- $[1 :: \text{True} :: \epsilon \mid \text{"5"} :: \epsilon] \text{Push True; } \epsilon \rightsquigarrow [\text{True} :: 1 :: \text{True} :: \epsilon \mid \text{"5"} :: \epsilon] \epsilon$

### 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. **POPEXCEPTION**: The stack is empty ( $S = \epsilon$ ).

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

$$\begin{array}{c} \text{POPSTACK} \\ \hline [c :: S \mid T] \text{ Pop}; P \rightsquigarrow [S \mid T] P \end{array} \qquad \begin{array}{c} \text{POPEXCEPTION} \\ \hline [\epsilon \mid T] \text{ Pop}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon \end{array}$$

Examples:

- $[1 :: \text{True} :: \epsilon \mid \text{"Unit"} :: \text{"False"} :: \epsilon] \text{ Pop}; \epsilon \rightsquigarrow [\text{True} :: \epsilon \mid \text{"Unit"} :: \text{"False"} :: \epsilon] \epsilon$
- $[\epsilon \mid \text{"5"} :: \epsilon] \text{ Pop}; \text{Push } 12; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"5"} :: \epsilon] \epsilon$

### 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 be prepended to the trace.

The **Trace** command has 1 fail state.

1. **TRACEEXCEPTION**: 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.

$$\begin{array}{c} \text{TRACESTACK} \\ \hline [c :: S \mid T] \text{ Trace}; P \rightsquigarrow [\text{Unit} :: S \mid \text{toString}(c) :: T] P \end{array} \qquad \begin{array}{c} \text{TRACEEXCEPTION} \\ \hline [\epsilon \mid T] \text{ Trace}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon \end{array}$$

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.

$$\text{toString}(123) = \text{"123"} \tag{1}$$

$$\text{toString}(\text{True}) = \text{"True"} \tag{2}$$

$$\text{toString}(\text{False}) = \text{"False"} \tag{3}$$

$$\text{toString}(\text{Unit}) = \text{"Unit"} \tag{4}$$

Examples:

- $[1 :: \text{True} :: \epsilon \mid \text{"Unit"} :: \text{"False"} :: \epsilon] \text{ Trace}; \epsilon \rightsquigarrow [\text{Unit} :: \text{True} :: \epsilon \mid \text{"1"} :: \text{"Unit"} :: \text{"False"} :: \epsilon] \epsilon$
- $[\epsilon \mid \text{"5"} :: \epsilon] \text{ Trace}; \text{Push } 12; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"5"} :: \epsilon] \epsilon$

### 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.

$$\begin{array}{c}
 \text{ADDSTACK} \\
 \hline
 i \text{ and } j \text{ are both integers} \\
 \hline
 [i :: j :: S \mid T] \text{ Add}; P \rightsquigarrow [(i + j) :: S \mid T] P
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ADDERROR1} \\
 \hline
 i \text{ or } j \text{ is not an integer} \\
 \hline
 [i :: j :: S \mid T] \text{ Add}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

$$\begin{array}{c}
 \text{ADDERROR2} \\
 \hline
 [\epsilon \mid T] \text{ Add}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ADDERROR3} \\
 \hline
 [c :: \epsilon \mid T] \text{ Add}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

Examples:

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

### 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.

$$\begin{array}{c}
 \text{SUBSTACK} \\
 \hline
 i \text{ and } j \text{ are both integers} \\
 \hline
 [i :: j :: S \mid T] \text{ Sub}; P \rightsquigarrow [(i - j) :: S \mid T] P
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SUBERROR1} \\
 \hline
 i \text{ or } j \text{ is not an integer} \\
 \hline
 [i :: j :: S \mid T] \text{ Sub}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

$$\begin{array}{c}
 \text{SUBERROR2} \\
 \hline
 [\epsilon \mid T] \text{ Sub}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}
 \qquad
 \begin{array}{c}
 \text{SUBERROR3} \\
 \hline
 [c :: \epsilon \mid T] \text{ Sub}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

Examples:

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

### 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.

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

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

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

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

Examples:

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

### 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.

$$\begin{array}{c}
 \text{DIVSTACK} \\
 \hline
 i \text{ and } j \text{ are both integers} \\
 \hline
 [i :: j :: S \mid T] \text{ Div}; P \rightsquigarrow [(i \div j) :: S \mid T] P
 \end{array}
 \qquad
 \begin{array}{c}
 \text{DIVERROR0} \\
 \hline
 i \text{ is an integer} \\
 \hline
 [i :: 0 :: S \mid T] \text{ Div}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

$$\begin{array}{c}
 \text{DIVERROR1} \\
 \hline
 i \text{ or } j \text{ is not an integer} \\
 \hline
 [i :: j :: S \mid T] \text{ Div}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}
 \qquad
 \begin{array}{c}
 \text{DIVERROR2} \\
 \hline
 [\epsilon \mid T] \text{ Div}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

$$\begin{array}{c}
 \text{DIVERROR3} \\
 \hline
 [c :: \epsilon \mid T] \text{ Div}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

Examples:

- $[16 :: 8 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Div}; \epsilon \rightsquigarrow [2 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[16 :: 0 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Div}; \text{Push Unit}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$
- $[16 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Div}; \text{Add}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$
- $[4 :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Div}; \text{Trace}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$

### 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.

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

$$\begin{array}{c}
 \text{ANDERROR2} \\
 \hline
 [\epsilon \mid T] \text{ And}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ANDERROR3} \\
 \hline
 [c :: \epsilon \mid T] \text{ And}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

Examples:

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

### 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.

$$\begin{array}{c}
 \text{ORSTACK} \\
 \hline
 a \text{ and } b \text{ are both booleans} \\
 \hline
 [a :: b :: S \mid T] \text{ Or}; P \rightsquigarrow [(a \vee b) :: S \mid T] P
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ORError1} \\
 \hline
 a \text{ or } b \text{ is not a boolean} \\
 \hline
 [a :: b :: S \mid T] \text{ Or}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

$$\begin{array}{c}
 \text{ORError2} \\
 \hline
 [\epsilon \mid T] \text{ Or}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}
 \qquad
 \begin{array}{c}
 \text{ORError3} \\
 \hline
 [c :: \epsilon \mid T] \text{ Or}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon
 \end{array}$$

Examples:

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



### 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.

$$\begin{array}{c}
 \text{NOTSTACK} \\
 \hline
 \frac{a \text{ is a boolean}}{[a :: S \mid T] \text{Not}; P \rightsquigarrow [(\neg a) :: S \mid T] P} \\
 \\
 \text{NOTERROR1} \\
 \hline
 \frac{a \text{ is not a boolean}}{[a :: S \mid T] \text{Not}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon} \\
 \\
 \text{NOTERROR2} \\
 \hline
 \frac{}{[\epsilon \mid T] \text{Not}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon}
 \end{array}$$

Examples:

- $[\text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{Not}; \epsilon \rightsquigarrow [\text{False} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[4 :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{Not}; \text{Pop}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$
- $[\epsilon \mid \text{"False"} :: \epsilon] \text{Not}; \text{Add}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$

### 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.

$$\begin{array}{c}
 \text{LTSTACK} \\
 \hline
 \frac{i \text{ and } j \text{ are both integers}}{[i :: j :: S \mid T] \text{Lt}; P \rightsquigarrow [(i < j) :: S \mid T] P} \\
 \\
 \text{LTERROR1} \\
 \hline
 \frac{i \text{ or } j \text{ is not an integer}}{[i :: j :: S \mid T] \text{Lt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon} \\
 \\
 \text{LTERROR2} \\
 \hline
 \frac{}{[\epsilon \mid T] \text{Lt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon} \\
 \\
 \text{LTERROR3} \\
 \hline
 \frac{}{[c :: \epsilon \mid T] \text{Lt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon}
 \end{array}$$

Examples:

- $[4 :: 5 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{Lt}; \epsilon \rightsquigarrow [\text{True} :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[5 :: 5 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{Lt}; \epsilon \rightsquigarrow [\text{False} :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[4 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{Lt}; \text{Trace}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$

### 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.

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

$$\frac{\text{GTERROR1} \quad i \text{ or } j \text{ is not an integer}}{[i :: j :: S \mid T] \text{ Gt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon}$$

$$\frac{\text{GTERROR2}}{[\epsilon \mid T] \text{ Gt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon}$$

$$\frac{\text{GTERROR3}}{[c :: \epsilon \mid T] \text{ Gt}; P \rightsquigarrow [\epsilon \mid \text{"Panic"} :: T] \epsilon}$$

Examples:

- $[4 :: 5 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Gt}; \epsilon \rightsquigarrow [\text{False} :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[10 :: 5 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Gt}; \epsilon \rightsquigarrow [\text{True} :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[5 :: 5 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Gt}; \epsilon \rightsquigarrow [\text{False} :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \epsilon$
- $[4 :: \text{True} :: \text{Unit} :: \epsilon \mid \text{"False"} :: \epsilon] \text{ Gt}; \text{Trace}; \epsilon \rightsquigarrow [\epsilon \mid \text{"Panic"} :: \text{"False"} :: \epsilon] \epsilon$

## 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"]