# CS320 Fall2023 Project Part 2

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 2, 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`

$\langle char \rangle$ ::= `a` | `b` | ... | `z` | $\langle digit \rangle$

$\langle sym \rangle$ ::= $\langle char \rangle$ | $\langle char \rangle\langle sym \rangle$

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

The $\langle char \rangle$ non-terminal is comprised of lowercase letters and digits.

### 2.2 Programs

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

$\langle com \rangle$ ::= `Push` $\langle const \rangle$ | `Pop` | `Swap` | `Trace`
 | `Add` | `Sub` | `Mul` | `Div`
 | `And` | `Or` | `Not`
 | `Lt` | `Gt`
 | `If` $\langle coms \rangle$ `Else` $\langle coms \rangle$ `End`
 | `Bind` | `Lookup`
 | `Fun` $\langle coms \rangle$ `End` | `Call` | `Return`

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

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 2, 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 Constants and Values

`Push` places constants, as defined by the grammar, onto the stack during evaluation. However, as we use more commands, we create things on the stack which are not simple constants. We generically refer to things that can go on the stack as *values*.

Constants are values. We have as constants: numbers, e.g. `-123`; booleans, e.g. `True`; unit, e.g. `Unit`; and symbols, e.g. `xyz123`. Symbols are new to part 2 and are used for names when we create bindings in the variable environment.

### 3.2 Configuration of Programs

A program configuration is of the following form.

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

- $S$: (**S**tack) stack of intermediate values
- $T$: (**T**race) list of strings logging the program trace
- $V$: (**V**ariable Environment) mapping from symbols to values (we will use the notation $x \rightarrowtail v$ to denote the map of the symbol $x$ to the value $v$)
- $P$: (**P**rogram) program commands to be interpreted

Examples:

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

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

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

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

### 3.3 Closures

In part 2, the Stack language will have functions. Functions will *not* be given as constants, but as commands including other blocks of commands, as already seen above in the grammar. To store functions on our stack we will have function values (called *closures*). These values relate to OCaml closure values (aka `fun ...`).

We will write closures as $\langle f, V, C\rangle$.

- $f$ is the symbol name of the closure. Unlike OCaml `fun`, our function values are *not* anonymous, they have names. Because they may be *recursive*.
- $V$ is a variable environment containing the variable binds *captured* by the closure, meaning they are those variables bound when the closure is created e.g. `let x = 3 in fun y -> x + y`. In the function, the variable `x` is *captured* since it is bound when the function is created.
- $C$ is the list of commands which make up the body of the function.

## 3.4 Program Reduction

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

$$[\,S_1 \mid T_1 \mid V_1\,]\ P_1 \rightsquigarrow [\,S_2 \mid T_2 \mid V_2\,]\ P_2$$

In one step, program configuration $[\,S_1 \mid T_1 \mid V_1\,]\ P_1$ evaluates to $[\,S_2 \mid T_2 \mid V_2\,]\ P_2$. 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.

## 3.5 Push

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

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

Examples:

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

## 3.6 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.

$$\frac{\textsc{PopStack}}{[\,c :: S \mid T \mid V\,]\ \texttt{Pop}; P \rightsquigarrow [\,S \mid T \mid V\,]\ P} \qquad \frac{\textsc{PopError}}{[\,\epsilon \mid T \mid V\,]\ \texttt{Pop}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

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

## 3.7 Swap

Given a stack of the form $c_1 :: c_2 :: S$ (constants $c_1$ and $c_2$ is on top of $S$), the `Swap` command changes the order of $c_1$ and $c_2$ and leaves the rest of stack $S$ unmodified.

The `Swap` command has 2 fail state.

1. SwapError1: The stack is empty ($S = \epsilon$).
2. SwapError3: The stack has only 1 element ($S = c :: \epsilon$).

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

$$\frac{\textsc{SwapStack}}{[\,c_1 :: c_2 :: S \mid T \mid V\,]\ \texttt{Swap}; P \rightsquigarrow [\,c_2 :: c_1 :: S \mid T \mid V\,]\ P} \qquad \frac{\textsc{SwapError1}}{[\,\epsilon \mid T \mid V\,]\ \texttt{Swap}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\frac{\textsc{SwapError2}}{[\,c :: \epsilon \mid T \mid V\,]\ \texttt{Swap}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

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

## 3.8 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.

$$\textsc{TraceStack}\quad \frac{}{[\, c :: S \mid T \mid V \,]\ \texttt{Trace}; P \rightsquigarrow [\, \texttt{Unit} :: S \mid \textit{toString}(c) :: T \mid V \,]\ P}$$

$$\textsc{TraceError}\quad \frac{}{[\, \epsilon \mid T \mid V \,]\ \texttt{Trace}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

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.

$$\textit{toString}(123) = \texttt{"123"} \tag{1}$$
$$\textit{toString}(\texttt{True}) = \texttt{"True"} \tag{2}$$
$$\textit{toString}(\texttt{False}) = \texttt{"False"} \tag{3}$$
$$\textit{toString}(\texttt{Unit}) = \texttt{"Unit"} \tag{4}$$
$$\textit{toString}(\texttt{abc}) = \texttt{"abc"} \tag{5}$$
$$\textit{toString}(\langle \texttt{abc}, V, P \rangle) = \texttt{"Fun<abc>"} \tag{6}$$

Examples:

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

## 3.9 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.

$$\textsc{AddStack}\quad \frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \mid V\,]\ \texttt{Add}; P \rightsquigarrow [\,(i+j) :: S \mid T \mid V\,]\ P}$$

$$\textsc{AddError1}\quad \frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \mid V\,]\ \texttt{Add}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\textsc{AddError2}\quad \frac{}{[\,\epsilon \mid T \mid V\,]\ \texttt{Add}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\textsc{AddError3}\quad \frac{}{[\,c :: \epsilon \mid T \mid V\,]\ \texttt{Add}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

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

## 3.10 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.

$$\textsc{SubStack}\quad \frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \mid V\,]\ \texttt{Sub}; P \rightsquigarrow [\,(i-j) :: S \mid T \mid V\,]\ P}$$

$$\textsc{SubError1}\quad \frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \mid V\,]\ \texttt{Sub}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\textsc{SubError2}\quad \frac{}{[\,\epsilon \mid T \mid V\,]\ \texttt{Sub}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\textsc{SubError3}\quad \frac{}{[\,c :: \epsilon \mid T \mid V\,]\ \texttt{Sub}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

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

## 3.11 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
$$\frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \mid V \,]\ \texttt{Mul}; P \rightsquigarrow [\, (i \times j) :: S \mid T \mid V \,]\ P}$$

MulError1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \mid V \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

MulError2
$$\frac{}{[\, \epsilon \mid T \mid V \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

MulError3
$$\frac{}{[\, c :: \epsilon \mid T \mid V \,]\ \texttt{Mul}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

Examples:

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

## 3.12 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
$$\frac{i \text{ and } j \text{ are both integers}, j \neq 0}{[\, i :: j :: S \mid T \mid V \,]\ \texttt{Div}; P \rightsquigarrow [\, (i \div j) :: S \mid T \mid V \,]\ P}$$

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

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

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

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

Examples:

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

## 3.13 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.

AndStack

$$\frac{a \text{ and } b \text{ are both booleans}}{[\, a :: b :: S \mid T \mid V \,]\ \texttt{And}; P \rightsquigarrow [\, (a \wedge b) :: S \mid T \mid V \,]\ P}$$

AndError1

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

AndError2

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

AndError3

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

Examples:

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

## 3.14 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.

OrStack

$$\frac{a \text{ and } b \text{ are both booleans}}{[\, a :: b :: S \mid T \mid V \,]\ \texttt{Or}; P \rightsquigarrow [\, (a \vee b) :: S \mid T \mid V \,]\ P}$$

OrError1

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

OrError2

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

OrError3

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

Examples:

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

## 3.15 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
$$\frac{a \text{ is a boolean}}{[\, a :: S \mid T \mid V \,]\ \texttt{Not}; P \rightsquigarrow [\, (\neg a) :: S \mid T \mid V \,]\ P}$$

NotError1
$$\frac{a \text{ is not a boolean}}{[\, a :: S \mid T \mid V \,]\ \texttt{Not}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

NotError2
$$\frac{}{[\, \epsilon \mid T \mid V \,]\ \texttt{Not}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

Examples:

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

## 3.16 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
$$\frac{i \text{ and } j \text{ are both integers}}{[\, i :: j :: S \mid T \mid V \,]\ \texttt{Lt}; P \rightsquigarrow [\, (i < j) :: S \mid T \mid V \,]\ P}$$

LtError1
$$\frac{i \text{ or } j \text{ is not an integer}}{[\, i :: j :: S \mid T \mid V \,]\ \texttt{Lt}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

LtError2
$$\frac{}{[\, \epsilon \mid T \mid V \,]\ \texttt{Lt}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

LtError3
$$\frac{}{[\, c :: \epsilon \mid T \mid V \,]\ \texttt{Lt}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

Examples:

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

## 3.17 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.

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

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

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

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

Examples:

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

### 3.18 If Else

Given a stack of the form $b :: S$ where $b$ is a boolean values, the `If` $C_1$ `Else` $C_2$ `End` command removes $b$ from the stack and evaluates $C_1$ if $b$ is `True` and $C_2$ if $b$ is `False` before continuing with the program.

The `If` $C_1$ `Else` $C_2$ `End` command has 2 fail states.

1. IfElseError1: $b$ is not a boolean.
2. IfElseError2: The stack is empty ($S = \epsilon$).

When `If` $C_1$ `Else` $C_2$ `End` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

$$\textsc{ThenStack}\quad \frac{}{[\,\texttt{True} :: S \mid T \mid V\,]\ \texttt{If}\ C_1\ \texttt{Else}\ C_2\ \texttt{End}; P \rightsquigarrow [\,S \mid T \mid V\,]\ C_1\ P}$$

$$\textsc{ElseStack}\quad \frac{}{[\,\texttt{False} :: S \mid T \mid V\,]\ \texttt{If}\ C_1\ \texttt{Else}\ C_2\ \texttt{End}; P \rightsquigarrow [\,S \mid T \mid V\,]\ C_2\ P}$$

$$\textsc{IfElseError1}\quad \frac{v \text{ is not a boolean}}{[\,v :: S \mid T \mid V\,]\ \texttt{If}\ C_1\ \texttt{Else}\ C_2\ \texttt{End}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

$$\textsc{IfElseError2}\quad \frac{}{[\,\epsilon \mid T \mid V\,]\ \texttt{If}\ C_1\ \texttt{Else}\ C_2\ \texttt{End}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

- $[\,\texttt{True} :: \epsilon \mid \texttt{"False"} :: \epsilon \mid \epsilon\,]\ \texttt{If Push 8;Trace; Else Trace; End;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"False"} :: \epsilon \mid \epsilon\,]\ \texttt{Push 8;Trace;}\epsilon$
- $[\,\texttt{False} :: \epsilon \mid \texttt{"False"} :: \epsilon \mid \epsilon\,]\ \texttt{If Push 8;Trace; Else Trace; End;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"False"} :: \epsilon \mid \epsilon\,]\ \texttt{Trace;}\epsilon$
- $[\,5 :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{If Push 3; Else Push 2; End;Add;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$
- $[\,\epsilon \mid \texttt{"4"} :: \epsilon \mid \epsilon\,]\ \texttt{If Push 3; Else Push 2; End;Add;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \texttt{"4"} :: \epsilon \mid \epsilon\,]\ \epsilon$

## 3.19 Bind

Given a stack of the form $x :: v :: S$ where $x$ is a symbol value and $v$ is any value, the `Bind` command removes $x$ and $v$ from the stack and assigns $x$ to $v$ in the variable environment.

The `Bind` command has 3 fail states.

1. BindError1: $x$ is not a symbol.
2. BindError2: The stack is empty ($S = \epsilon$).
3. BindError3: The stack has only 1 element ($S = c :: \epsilon$).

When `Bind` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates. As mentioned before, the notation $x \rightarrowtail v$ in the rules below denotes the map of the symbol $x$ to the value $v$.

BindStack

$$\frac{x \text{ is a symbol}}{[\, x :: v :: S \mid T \mid V \,]\ \texttt{Bind}; P \rightsquigarrow [\, S \mid T \mid x \rightarrowtail v :: V \,]\ P}$$

BindError1

$$\frac{x \text{ is not a symbol}}{[\, x :: v :: S \mid T \mid V \,]\ \texttt{Bind}; P \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

BindError2

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

BindError3

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

Examples:

- $[\, \texttt{x} :: \texttt{True} :: \epsilon \mid \epsilon \mid \epsilon \,]\ \texttt{Bind}; \epsilon \rightsquigarrow [\, \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon \,]\ \epsilon$
- $[\, \texttt{True} :: \texttt{x} :: \epsilon \mid \epsilon \mid \epsilon \,]\ \texttt{Bind}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon \,]\ \epsilon$
- $[\, \texttt{x} :: \epsilon \mid \epsilon \mid \epsilon \,]\ \texttt{Bind}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon \,]\ \epsilon$
- $[\, \epsilon \mid \texttt{"4"} :: \epsilon \mid \epsilon \,]\ \texttt{Bind}; \epsilon \rightsquigarrow [\, \epsilon \mid \texttt{"Panic"} :: \texttt{"4"} :: \epsilon \mid \epsilon \,]\ \epsilon$

## 3.20 Lookup

Given a stack of the form $x :: S$ where $x$ is a symbol value, the `Lookup` command removes $x$ from the stack, finds the *most recent* value it is bound to, and puts this value on the stack.

The `Lookup` command has 3 fail states.

1. LookupError1: $x$ is not a symbol.
2. LookupError2: The stack is empty ($S = \epsilon$).
3. LookupError3: $x$ is not bound to a value.

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

$$\textsc{LookupStack}\quad \frac{x \text{ is a symbol} \qquad x \rightarrowtail v \in V}{[\, x :: S \mid T \mid V \,]\ \texttt{Lookup};P \leadsto [\, v :: S \mid T \mid V \,]\ P}$$

$$\textsc{LookupError1}\quad \frac{x \text{ is not a symbol}}{[\, x :: S \mid T \mid V \,]\ \texttt{Lookup};P \leadsto [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

$$\textsc{LookupError2}\quad \frac{}{[\, \epsilon \mid T \mid V \,]\ \texttt{Lookup};P \leadsto [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

$$\textsc{LookupError3}\quad \frac{x \rightarrowtail v \notin V}{[\, x :: \epsilon \mid T \mid V \,]\ \texttt{Lookup};P \leadsto [\, \epsilon \mid \texttt{"Panic"} :: T \mid V \,]\ \epsilon}$$

Examples:

- $[\, \texttt{x} :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon \,]\ \texttt{Lookup};\epsilon \leadsto [\, \texttt{True} :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon \,]\ \epsilon$
- $[\, \texttt{x} :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \texttt{x} \rightarrowtail \texttt{False} :: \epsilon \,]\ \texttt{Lookup};\epsilon \leadsto [\, \texttt{True} :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \texttt{x} \rightarrowtail \texttt{False} :: \epsilon \,]\ \epsilon$
- $[\, \texttt{Unit} :: \epsilon \mid \epsilon \mid \epsilon \,]\ \texttt{Lookup};\epsilon \leadsto [\, \epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon \,]\ \epsilon$
- $[\, \epsilon \mid \epsilon \mid \epsilon \,]\ \texttt{Lookup};\epsilon \leadsto [\, \epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon \,]\ \epsilon$
- $[\, \texttt{x} :: \epsilon \mid \epsilon \mid \texttt{y} \rightarrowtail \texttt{True} :: \texttt{y} \rightarrowtail \texttt{False} :: \epsilon \,]\ \texttt{Lookup};\epsilon \leadsto [\, \epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon \,]\ \epsilon$

## 3.21 Fun

See Section 4 for a detailed example using function definition, function call and function return.

`Fun` $C$ `End` is a command representing a function definition. Given a stack of the form $x :: S$ where $x$ is a symbol value, the `Fun` $C$ `End` command creates a closure value (explained in Section 3.3) on the stack with name $x$, the current variable environment, and the commands $C$ defining the function.

The `Fun` $C$ `End` command has 2 fail states.

1. FunError1: $x$ is not a symbol.
2. FunError2: The stack is empty ($S = \epsilon$).

When `Fun` $C$ `End` fails, the stack is cleared, the string `"Panic"` is prepended to the trace and the program terminates.

FunStack
$$\frac{x \text{ is a symbol}}{[\,x :: S \mid T \mid V\,]\ \texttt{Fun}\ C\ \texttt{End};P \rightsquigarrow [\,\langle x, V, C\rangle :: S \mid T \mid V\,]\ P}$$

FunError1
$$\frac{x \text{ is not a symbol}}{[\,x :: S \mid T \mid V\,]\ \texttt{Fun}\ C\ \texttt{End};P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

FunError2
$$\frac{}{[\,\epsilon \mid T \mid V\,]\ \texttt{Fun}\ C\ \texttt{End};P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

- $[\,\texttt{f} :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon\,]\ \texttt{Fun Trace; End;}\epsilon \rightsquigarrow [\,\langle \texttt{f}, \texttt{x} \rightarrowtail \texttt{True} :: \epsilon, \texttt{Trace;}\rangle :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon\,]\ \epsilon$
- $[\,\texttt{True} :: 2 :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Fun Push 3; End;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$
- $[\,\epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Fun Gt; End;}\epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$

## 3.22 Call

See Section 4 for a detailed example using function definition, function call and function return.

`Call` is used to implement the call of a (potentially recursive) function. Given a stack of the form $\langle f, V, C\rangle :: a :: S$ where $\langle f, V, C\rangle$ is a closure value (explained in Section 3.3), the `Call` command consumes $\langle f, V, C\rangle$ and $a$ , then executes the commands $C$ from the closure with variable environment $f \rightarrowtail \langle f, V, C\rangle :: V$ (notice that this is the environment from the closure updated with a new map associating the symbol $f$ with the closure itself - this is needed for recursive functions). The command also need to update the stack to have the current continuation and the value $a$. The current continuation is itself a closure $\langle \texttt{cc}, V', P'\rangle$ where `cc` is just a symbol standing for current continuation, V' is the current environment and $P'$ is the list of remaining commands, put onto the stack.

The `Call` command has 2 fail states.

1. CallError1: $v$ is not a closure value.
2. CallError2: The stack is empty ($S = \epsilon$).
3. CallError3: The stack has only 1 element ($S = c :: \epsilon$).

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

CallStack

$$\frac{}{\begin{array}{l}[\,\langle f, V_f, C\rangle :: a :: S \mid T \mid V\,]\ \texttt{Call}; P \\ \quad\rightsquigarrow [\,a :: \langle \texttt{cc}, V, P\rangle :: S \mid T \mid f \rightarrowtail \langle f, V_f, C\rangle :: V_f\,]\ C\end{array}}$$

CallError1

$$\frac{c \text{ is not a closure value.}}{[\,c :: S \mid T \mid V\,]\ \texttt{Call}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

CallError2

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

CallError3

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

Examples:

- $$\begin{array}{l}[\,\langle \texttt{f}, \texttt{x} \rightarrowtail \texttt{True}, \texttt{Trace}; \epsilon\rangle :: 2 :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{False} :: \epsilon\,]\ \texttt{Call}; \epsilon \\ \rightsquigarrow \\ [\,2 :: \langle \texttt{cc}, \texttt{x} \rightarrowtail \texttt{False} :: \epsilon, \epsilon\rangle :: \epsilon \mid \epsilon \mid \texttt{f} \rightarrowtail \langle \texttt{f}, \texttt{x} \rightarrowtail \texttt{True}, \texttt{Trace}; \epsilon\rangle :: \texttt{x} \rightarrowtail \texttt{True} :: \epsilon\,]\ \texttt{Trace}; \epsilon\end{array}$$
- $[\,\texttt{True} :: 2 :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Call}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$
- $[\,\langle \texttt{f}, \epsilon, \epsilon\rangle :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Call}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$

## 3.23 Return

See Section 4 for a detailed example using function definition, function call and function return.

`Return` is used to implement the return call from a function. The behavior of `Return` is similar to the one of `Call`, but no continuation is pushed and no name is bound.
Given a stack of the form $v :: a :: S$ where $v$ is a closure value, $v = \langle f, V, C\rangle$, the `Return` command consumes $v$, then executes the commands $C$ with variable environment $V$.

The `Return` command has 2 fail states.

1. ReturnError1: $v$ is not a closure value.
2. ReturnError2: The stack is empty ($S = \epsilon$).
3. ReturnError3: The stack has only 1 element ($S = c :: \epsilon$).

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

ReturnStack
$$\frac{}{[\,\langle f, V_f, C\rangle :: a :: S \mid T \mid V\,]\ \texttt{Return}; P \rightsquigarrow [\,a :: S \mid T \mid V_f\,]\ C}$$

ReturnError1
$$\frac{c \text{ is not a closure value.}}{[\,c :: a :: S \mid T \mid V\,]\ \texttt{Return}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

ReturnError3
$$\frac{}{[\,\epsilon \mid T \mid V\,]\ \texttt{Return}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

ReturnError3
$$\frac{}{[\,c :: \epsilon \mid T \mid V\,]\ \texttt{Return}; P \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: T \mid V\,]\ \epsilon}$$

Examples:

- $[\,\langle \texttt{f}, \texttt{x} \rightarrowtail \texttt{True}, \texttt{Trace}; \epsilon\rangle :: 2 :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Return}; \epsilon \rightsquigarrow [\,2 :: \epsilon \mid \epsilon \mid \texttt{x} \rightarrowtail \texttt{True} :: \epsilon\,]\ \texttt{Trace}; \epsilon$
- $[\,\texttt{True} :: 2 :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Return}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$
- $[\,\langle \texttt{f}, \epsilon, \epsilon\rangle :: \epsilon \mid \epsilon \mid \epsilon\,]\ \texttt{Return}; \epsilon \rightsquigarrow [\,\epsilon \mid \texttt{"Panic"} :: \epsilon \mid \epsilon\,]\ \epsilon$

## 4 Function Example

Functions in the Stack language are based on continuation-passing style (CPS). This means, that when we call a function, on top of passing arguments to it, we must also pass to it a function representing the continuation which will need to be executed when the function call terminates. This means that we have *two* ways of invoking functions: `Call` and `Return`. The first is a normal function call, where the function we call may recursively use itself and it results in a value. `Return` is a call to a function which is represented by a closure on the stack and it can be used to call the continuation which will resume the execution of the caller. The names follow our intuition from other languages, as unlike `yield` in Python where the function may resume, `return` does not allow resumption of a function.

Consider the OCaml program for computing factorial:

```
let rec factorial = fun n ->
   if 2 > n then
     1
   else
     factorial(n - 1) * n

let _ = print_int(factorial(4))
```

And the equivalent Stack language program:

```
Push factorial;
Fun
  Push n;
  Bind;

  Push 2;
  Push n;
  Lookup;
  Gt;

  If
    Push 1;
    Swap;
    Return;
  Else
    Push n;
    Lookup;
    Push -1;
    Add;

    Push factorial;
    Lookup;
    Call;

    Push n;
    Lookup;
    Mul;

    Swap;
    Return;
  End;
```

```
End;

Push factorial;
Bind;

Push 4;
Push factorial;
Lookup;

Call;

Trace;
```

Although the programs differ in visual complexity, they work in a similar way:

-
  ```
  Push factorial;
  Fun ...
  ```

  As with `let rec factorial = fun n -> ...`, we are creating a recursive function with the name `factorial` and taking a single argument.

-
  ```
  ...
  Fun
    Push n;
    Bind;
    ...
  ```

  Implicit in the OCaml program, our argument in the Stack language is unnamed to start, so we assign it the name `n`.

-
  ```
  ...
  Push 2;
  Push n;
  Lookup;
  Gt;
  ...
  ```

  Just `2 > n` directly.

-
  ```
  ...
  If
    Push 1;
    Swap;
    Return;
  ...
  ```

  `if 2 > n then 1`. When `factorial` is called, it expects a continuation (a function) on its stack before its arguments. This continuation takes our result as an argument. Since arguments are put *before* a function on the stack, we must use `Swap`, and then we invoke the continuation with `Return`.

-
  ```
  ...
  Else
    Push n;
    Lookup;
    Push -1;
    Add;
  ```

```
      Push factorial;
      Lookup;
      Call;

      Push n;
      Lookup;
      Mul;

      Swap;
      Return;
    End;
    ...
```

`else factorial(n - 1) * n`. Compute our argument to the recursive call. `Lookup` ourselves. `Call` recursively. The result of the recursive call is placed on our stack. We may then do our multiplication and `Return` our result.

-
  ```
  ....
  Push factorial;
  Bind;
  ...
  ```

  As with our argument `n` in our function, this task is handled in OCaml during `let rec factorial = fun n -> ....` Using `Fun ... End` in Stack *just* makes a closure and *places it on the stack*. To have it be bound to a name to lookup, we must explicitly `Bind`.

-
  ```
  Push 4;
  Push factorial;
  Lookup;

  Call;

  Trace;
  ```

  `print_int(factorial(4))`. As we have already seen in the recursive uses of `fibonacci`, we call a function by pushing our argument, the function, and then using `Call`. After we get back our result, it will be on our stack, so we may use `Trace` to print it out.

## 5 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"]`

- Factorial from 4.

  ```
  ...
  ```

  Result: `Some ["24"]`

- Define a function for polynomial $x^2 - 4x + 7$ and evaluate at 3:

```
Push poly;
Fun
  Push x;
  Bind;

  Push x;
  Lookup;
  Push x;
  Lookup;
  Mul;

  Push -4;
  Push x;
  Lookup;
  Mul;

  Add;

  Push 7;
  Add;

  Return;
End;

Push 3;
Swap;
Call;

Trace;
```

Result: `Some ["4"]`