Stack Language

The stack-oriented programming language described below is used for our interpreter as well as the low-level language for our compiler. Value (constants), will be moved onto the stack and manipulated to produce an output which can go to Trace.

Our Stack Program consists of three pieces of memory:

[S | T | V] P

- S (Stack): Stack of intermediate values
- T (Trace): String list storing all calls to the 'Trace' command
- V: (Variable Environment): Mapping symbols to constants of any type
  (x → v, denotes Symbol x corresponds to constant v)
- P (Program): Remaining program to be interpreted

Constants

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<nat> ::= <digit> | <digit><nat>
<int> ::= <nat> | -<nat>
<bool> ::= True | False
<char> ::= a | b | c | d | … | z
<char> ::= A | B | C | D | … | Z
<symbol> ::= <char> | <char><symbol>
<const> ::= <int> | <bool> | <symbol> | Unit

Programs

<com> ::=
|Push <const> | Dup | Pop
| Add | Sub | Mul | Div
| Swap | Over
| And | Or | Not
| Lt | Gt | Eq
| If <prog> Else <prog> End
| Call | Ret
| Bind | Lookup
| Fun <prog> End
| Trace

<prog> ::= ϵ | <com>;<prog>

Single Step Reduction:

The following descriptions for each command follow Single Step Reduction
The ' $\rightarrow$ ' symbol denotes one step, unless a natural, k, follows it as such ' $\rightarrow^k$ '

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

Push

Given the command ‘Push c,’ where c is any constant, Push will prepend c onto the stack.

Fail States:

Push never fails

Push

[S | T | V] Push c; P → [c :: S | T | V] P

Dup

Given a stack with at least one constant c_1 on top, Dup will copy c_1 and push c_2 onto the stack.

Fail States:

1. Dup fails if the stack is empty

Dup

[c_1 :: S | T | V] Dup; P → [c_2 :: c_1 :: S | T | V] P

Dup Failure

[ϵ | T | V] Dup; P → "Dup failure. Empty stack. Nothing to Duplicate"

Pop

Given a stack with at least one constant c on top, Pop will remove c from the stack.

Fail States:

1. Pop fails if the stack is empty

Pop

[c :: S | T | V] Pop; P → [S | T | V] P

PopFailure

[ϵ | T | V] Pop; P → "Pop failure. Empty stack. Nothing to Pop"

Add

Given a stack of the form x :: y :: S where x and y are integer values. Add removes x and y from the stack and pushes their sum (x+y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Add

[x :: y :: S | T | V] Add; P → [(x+y) :: S | T | V] P

Add Error 1

[x :: y :: S | T | V] Add; P → “Add failure. Requires two integers”

Add Error 2

[ϵ | T | V] Add; P → “Add failure. Empty stack. Nothing to Add”

Add Error 3

[x :: ϵ | T | V] Add; P → “Add failure. Only one element on stack. Requires two integers”

Sub

Given a stack of the form x :: y :: S where x and y are integer values. Sub removes x and y from the stack and pushes their difference (x-y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Sub

[x :: y :: S | T | V] Sub; P → [(x-y) :: S | T | V] P

Sub Error 1

[x :: y :: S | T | V] Sub; P → “Sub failure. Requires two integers”

Sub Error 2

[ϵ | T | V] Sub; P → “Sub failure. Empty stack. Nothing to Sub”

Sub Error 3

[x :: ϵ | T | V] Sub; P → “Sub failure. Only one element on stack. Requires two integers”

Mul

Given a stack of the form x :: y :: S where x and y are integer values. Mul removes x and y from the stack and pushes their product (x*y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Mul

[x :: y :: S | T | V] Mul; P → [(x*y) :: S | T | V] P

Mul Error 1

[x :: y :: S | T | V] Mul; P → “Mul failure. Requires two integers”

Mul Error 2

[ϵ | T | V] Mul; P → “Mul failure. Empty stack. Nothing to Mul”

Mul Error 3

[x :: ϵ | T | V] Mul; P → “Mul failure. Only one element on stack. Requires two integers”

Div

Given a stack of the form x :: y :: S where x and y are integer values. Div removes x and y from the stack and pushes their quotient (x/y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Div

[x :: y :: S | T | V] Div; P → [(x/y) :: S | T | V] P

Div Error 1

[x :: y :: S | T | V] Div; P → “Div failure. Requires two integers”

Div Error 2

[ϵ | T | V] Div; P → “Div failure. Empty stack. Nothing to Div”

Div Error 3

[x :: ϵ | T | V] Div; P → “Div failure. Only one element on stack. Requires two integers”

Swap

Given a stack of the form x :: y :: S, where x and y are any constant on top of the stack. Swap changes the order of the two elements and leaves the rest of the stack unchanged.

Fail States:

1. Stack is empty
2. Stack only has one element

Swap

[x :: y :: S | T | V] Swap; P → [y :: x :: S | T | V] P

Swap Error 1

[ϵ :: S | T | V] Swap; P → “Swap failure. Empty stack. Nothing to Swap”

Swap Error 2

[x :: ϵ | T | V] Swap; P → “Swap failure. Two constants do not exist at the top of the stack”

Over

Given a stack of the form x :: y :: S, where x and y are any constant on top of the stack. Over copies y, and pushes it to the top of the stack.

Fail States:

1. Stack is empty
2. Stack only has one element

Over

[x :: y :: S | T | V] Over; P → [y :: x :: y :: S | T | V] P

Over Error 1

[ϵ :: S | T | V] Over; P → “Over failure. Empty stack. Nothing to Over”

Over Error 2

[x :: ϵ | T | V] Over; P → “Over failure. Two constants do not exist at the top of the stack”

And

Given a stack of the form a :: b :: S, where a and b are boolean values. And removes a and b from the stack and pushes their conjunction (a ^ b) onto the stack.

Fail States:

1. Either a or b is not a boolean
2. Stack is empty
3. Stack only has one element

And

[a :: b :: S | T | V] And; P → [(x ^ y) :: S | T | V] P

And Error 1

[a :: b :: S | T | V] And; P → “And failure. Requires two booleans”

And Error 2

[ϵ | T | V] And; P → “And failure. Empty stack. Nothing to And”

And Error 3

[a :: ϵ | T | V] And; P → “And failure. Only one element on stack. Requires two booleans”

Or

Given a stack of the form a :: b :: S, where a and b are boolean values. Or removes a and b from the stack and pushes their disjunction (a v b) onto the stack.

Fail States:

1. Either a or b is not a boolean
2. Stack is empty
3. Stack only has one element

Or

[a :: b :: S | T | V] Or; P → [(x v y) :: S | T | V] P

Or Error 1

[a :: b :: S | T | V] Or; P → “Or failure. Requires two booleans”

Or Error 2

[ϵ | T | V] Or; P → “Or failure. Empty stack. Nothing to Or”

Or Error 3

[a :: ϵ | T | V] Or; P → “Or failure. Only one element on stack. Requires two booleans”

Not

Given a boolean c on top of the stack, Not removes c from the stack and pushes it's opposite value (¬c) onto the stack.

Fail States:

1. c is not a boolean
2. Stack is empty

Not

[c :: S | T | V] Not; P → [(¬c) :: S | T | V] P

Not Error 1

[c :: S | T | V] Not; P → "Not failure. Top of stack must be a boolean"

Not Error 2

[ϵ | T | V] Not; P → "Not failure. Empty stack. Nothing to Not"

Lt

Given a stack of the form x :: y :: S where x and y are integer values. Lt removes x and y from the stack and pushes their comparison (x < y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Lt

[x :: y :: S | T | V] Lt; P → [(x < y) :: S | T | V] P

Lt Error 1

[x :: y :: S | T | V] Lt; P → "Lt failure. Requires two integers"

Lt Error 2

[ϵ | T | V] Lt; P → "Lt failure. Empty stack. Nothing to Lt"

Lt Error 3

[x :: ϵ | T | V] Lt; P → "Lt failure. Only one element on stack. Requires two integers"

Gt

Given a stack of the form x :: y :: S where x and y are integer values. Gt removes x and y from the stack and pushes their comparison (x > y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Gt

[x :: y :: S | T | V] Gt; P → [(x > y) :: S | T | V] P

Gt Error 1

[x :: y :: S | T | V] Gt; P → “Gt failure. Requires two integers”

Gt Error 2

[ϵ | T | V] Gt; P → “Gt failure. Empty stack. Nothing to Gt”

Gt Error 3

[x :: ϵ | T | V] Gt; P → “Gt failure. Only one element on stack. Requires two integers”

Eq

Given a stack of the form x :: y :: S where x and y are integer values. Eq removes x and y from the stack and pushes their comparison (x = y) onto the stack.

Fail States:

1. Either x or y is not an integer
2. Stack is empty
3. Stack only has one element

Eq

[x :: y :: S | T | V] Eq; P → [(x = y) :: S | T | V] P

Eq Error 1

[x :: y :: S | T | V] Eq; P → “Eq failure. Requires two integers”

Eq Error 2

[ϵ | T | V] Eq; P → “Eq failure. Empty stack. Nothing to Eq”

Eq Error 3

[x :: ϵ | T | V] Eq; P → “Eq failure. Only one element on stack. Requires two integers”

If Statement

Given a boolean c on top of the stack, the If command removes c, and depending on it's value will run either the main branch if c is true, or the else branch if c is false.

Fail States:

1. c is not a boolean
2. Stack is empty

If-Main

[True :: S | T | V] If C_1 Else C_2 End; P → [S | T | V] C_1; P

If-Else

[False :: S | T | V] If C_1 Else C_2 End; P → [S | T | V] C_2; P

If Error 1

[c :: S | T | V] If C_1 Else C_2 End; P → "If failure. Top of stack must be a boolean"

If Error 2

[ϵ | T | V] If C_1 Else C_2 End; P → "If failure. Empty stack. Nothing to If"

Bind

Given a stack n :: x, where n is a symbol and x is any constant, the Bind command removes n and x from the stack, and maps n to x in the variable environment.

Fail States:

1. n is not a symbol
2. Stack is empty
3. Stack only has one element

Bind

[n :: x :: S | T | V] Bind; P → [S | T | (n → x)] P

Bind Error 1

[c :: x :: S | T | V] Bind; P → "Bind failure. Requires top element to be symbol"

Bind Error 2

[ϵ | T | V] Bind; P → "Bind failure. Empty stack. Nothing to Bind"

Eq Error 3

[c :: ϵ | T | V] Bind; P → "Bind failure. Only one element on stack. Requires a symbol preceding any constant"

Lookup

Given a stack with the symbol n on top, Lookup removes n from the stack and pushes the most recent constant it is bound to in the variable environment.

Fail States:

1. n is not a symbol
2. Stack is empty
3. n is not bound to any constant in the variable environment

Lookup

[n :: S | T | (n → x) :: V] Lookup; P → [x :: S | T | (n → x) :: V] P

Lookup Error 1

[n :: S | T | V] Lookup; P → “Lookup failure. Requires top element to be symbol”

Lookup Error 2

[ϵ | T | V] Lookup; P → “Lookup failure. Empty stack. Nothing to Lookup”

Eq Error 3

[n :: S | T | V] Lookup; P → “Lookup failure. Symbol is not bound to any variable”

Fun

Fun P_1 End represents a function definition. Given a stack of the form x :: S where x is a symbol, the Fun P_1 End command removes x and pushes a closure constant on the stack with name x, the current variable environment, and the subprogram P_1 defining the function.

Fail States:

1. x is not a symbol
2. Stack is empty

Fun

[x :: S | T | V] Fun P_1 End; P → [<x, C, P_1> :: S | T | V] P

Fun Error 1

[x :: S | T | V] Fun P_1 End; P → “Fun failure. Requires top element to be symbol”

Fun Error 2

[ϵ | T | V] Fun P_1 End; P → “Fun failure. Empty stack. Nothing to Fun”

Call

Given a stack of the form <n, V, C> :: a :: S, where <n, V, C> is a closure constant, Call consumes the closure <n, V, C> and a, then executes the commands C from the closure with the variable environment n → <n, V, C> :: V *.

Call also updates the stack to have the current continuation and the value a. The current continuation is itself a closure ⟨cc, V ′, P ′⟩ where cc is just a symbol standing for current continuation, V ' is the current environment and P ' is the list of remaining commands, pushed onto the stack.

Fail States:

1. Top value is not a closure
2. Stack is empty
3. Stack only has one element

Call

[<n, V, C> :: a :: S | T | V] Call; P →
[a :: <cc, V, P> :: S | T | (n → <n, V_n, C>) :: V_n] C

Call Error 1

[x :: S | T | V] Call; P → "Call failure. Top element is not closure"

Call Error 2

[ϵ | T | V] Call; P → "Call failure. Empty stack. Nothing to Call"

Call Error 3

[x :: ϵ | T | V] Call; P → "Call failure. Requires closure as top element, followed by some constant"

Ret

Given a stack of the form v :: a :: S, where v is some closure value (<f,V,C>) and a is some constant, Ret consumes v and executes the commands C with variable environment V.

Fail States:

1. v is not a closure
2. Stack is empty
3. Stack only has one element

Ret

[<n, V_n, C> :: a :: S | T | V] Ret; P → [a :: S | T | V_n] C

Ret Error 1

[x :: S | T | V] Ret; P → “Ret failure. Top element is not closure”

Ret Error 2

[ϵ | T | V] Ret; P → “Ret failure. Empty stack. Nothing to Ret”

Ret Error 3

[x :: ϵ | T | V] Ret; P → “Ret failure. Requires closure as top element, followed by some constant”

Function Examples:

Functions in stack-language

Upon calling 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 leaves two ways of invoking functions:

- Call: Normal function call where the function we call may recursively use itself and it results in a value
- Return: 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

Absolute Value

```
Push abs;
Fun
   Push n;
   Bind;

   Push n;
   Lookup;

   Push 0;
   Swap;
   Lt;
   If
         Push n;
         Lookup;
         Push -1;
         Mul;
   Else
         Push n;
         Lookup;
   End;
   Swap;          | Must be done before returning so closure if top value on stack
   Ret;           | Eliminates closure from stack, leaving only the return value
End;

Push abs;   |
Bind;       |   Binds the function defined above to symbol abs
Push -2;    |
Push abs;   |
Lookup;     |
Call;       |   Calls abs with -2 as parameter

Trace;      |   Prints output when function returns value
```

Factorial

```
Push factorial;
Fun
        Push n;
        Bind;
        Push n;
        Lookup;
        Push 2;
        Gt;
        If
                Push 1;
                Swap;
                Ret;
        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;
```

Power

```
Push pow;
Fun
  Push exponent;
  Bind;

  Swap;
  Push base;
  Bind;

  Push exponent;
  Lookup;
  Push 1;
  Gt;
  If
        Push 1;
        Swap;
        Ret;
  Else
        Push base;
        Lookup;
        Push exponent;
        Lookup;
        Push 1;
        Swap;
        Sub;

        Push pow;
        Lookup;
        Call;
        Push base;
        Lookup;
        Mul;
        Swap;
        Ret;          | Remove closure and execute the commands from Push base, down until next return
  End;
End;

Push pow;
Bind;

Push 4;               | Base
Push 3;               | Exponent
```

```
Push pow;
Lookup;
Call;                | Call on the function we have stored in the variable environment
Trace;               | Output should return 64
```