Design Document

Ben Geyer (geyerb), Mateo Rey-Rosa (reyrosam), Ryan Chesla (cheslar)

1. Introduction

What is the name of your language?

StackyStack

What is the language's paradigm?

Stack-based

2. Design

What *features* does your language include? Be clear about how you satisfied the constraints of the feature menu.

Basic data types

Details:

- Integers/Floats
- Booleans

Integers and Floats are implemented at the core, while booleans are implemented via syntactic sugar using the integer values of 0 and 1. Integers and Floats will both be the same type in the core, however methods will be created using syntactic sugar to differentiate the two where necessary (such as integer division). We decided to move the Boolean values and some Integer/Float operations out of the core to declutter the core and remove redundancy as these values can all be represented by basic numbers, which are already defined in the core.

Arithmetic Operations

Details:

- Multiplication
- Power

- Division
- Integer Division
- Modulus
- Subtraction
- Add
- AddOne
- SubtractOne

Multiplication, addition, and powers are implemented at the core as can be seen from the concrete syntax of the language. We chose these to be at the core because the other operations can be implemented with sugar using these building blocks. To make their use more accessible and to make the language appear more uniform, syntactic sugar is used to push the appropriate expression for the user when the respective command is used. Division can be created using syntactic sugar by combining both the -1 power and multiplication. Integer division and modulus can be implemented using special stack operations. Subtraction is implemented via syntactic sugar by creating a program that contains both negation and addition. Division Both AddOne and SubtractOne are used by the looping mechanisms, and merely add or subtract one from the counter also stored in the program. Using the block feature, these programs are then created into one command.

Logic Operations

Details:

- Greater Than/Greater than or equal
- Less Than/Less than or equal
- And
- Or
- Equal
- Negation
- Not

Only Less than and Equal are implemented at the core, as the other operations can be created by combining types defined at the core. Negation is implemented by multiplying the newInt value by -1 and passing this to block. Not is created by pushing both newFalse and equ to the stack. And and Or both use If-then-else statements to determine the result of the expressions.

Conditionals

Details:

- If-then-else statements

The if-then-else statements are implemented at the core and take in two programs. They will use the value at the top of the stack to determine which program to run. This value can either be an integer or another stack. An integer value of zero or an empty stack will be evaluated as a boolean false, otherwise it will evaluate as true. If there is no value at the top of the stack, the “else” program will be run.

Recursion/Loops

Details:

- While loop
- For Loop
- Foreach Loop
- Reverse
- Recursion

The for loop and the foreach loops are based off of the while loop in the core. Reverse is used to implement the foreach loop, as well as several stack operations including extend. Recursion can also be used by combining several of the core features and the expression data type. Using the IfElse feature we can declare the base case and the recursive case. We chose to keep the while loop at the core as looping is necessary but the other loops can be represented using a while loop and other core types.

Procedures/functions with arguments

Details:

- Functions with arguments
- First class functions

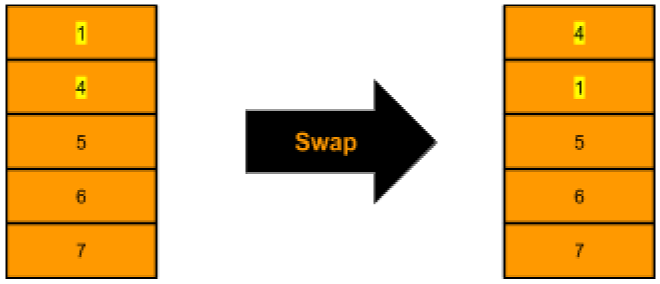
Functions are implemented at the core and essentially are mapped from a string to a program which can be used to declare and later call the functions using the cmd data type. Functions themselves can also be pushed on the stack, allowing them to be used as arguments for other functions. To call them, another function can use the “Exec” operation, which is also defined at the core of the language. This allows the use of first class functions, one of the features from the feature menu. We chose to keep functions at the core of the language as they must access a special state of the program outside the stack to map their name to the program they execute.

Stack Operations

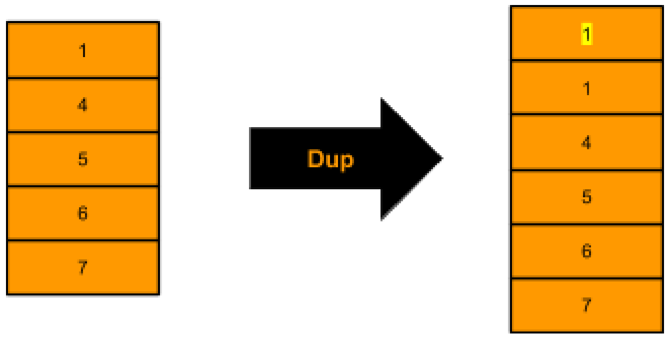
Details:

Core Stack Operations

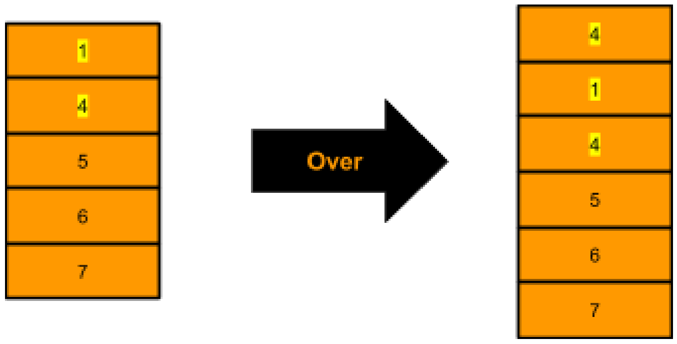
- NewStack: creates an “inner stack” -- a stack inside the current stack.
- RunInside: takes in a program and runs the program using the inner stack on the top of the stack.
- Insert: inserts the first value on the outer stack into an inner stack that must be the second value on the outer stack.
- Extract: removes the first element from the inner stack and puts it at the front of the outer stack.
- Swap: reverse top two stack items.



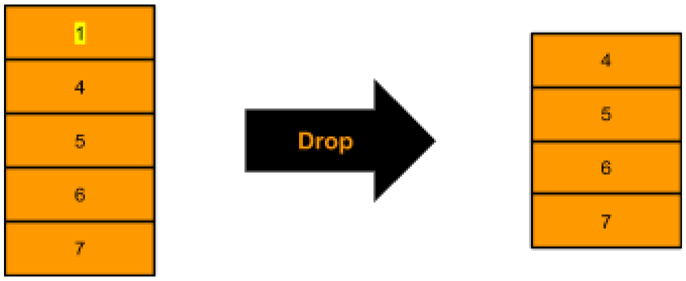
- Dup: duplicates the top stack item.



- Over: copies second item to top.



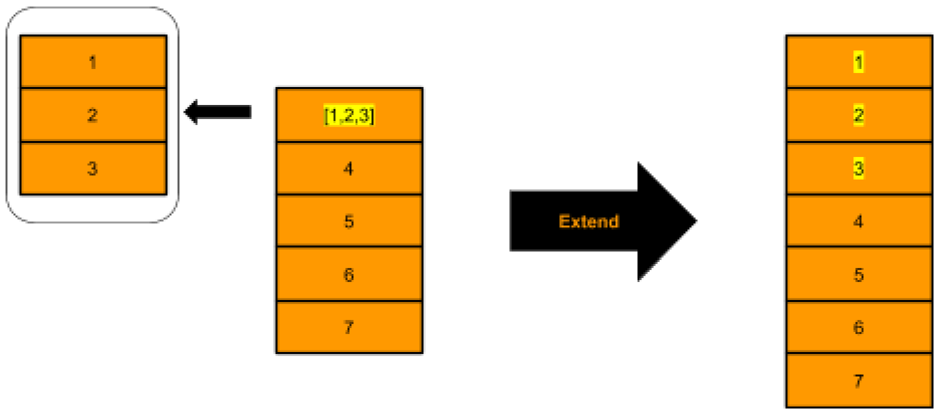
- Drop: discards the top stack item.



The operations listed above were implemented at the core because they are the most important for implementing other stack operations using syntactic sugar, and many of them would be impossible to create outside the core. From the set of core stack manipulation features, more robust versions of these operations can be implemented in syntactic sugar, which allows the operations to be represented by a program of stack operations. Both extend and compress are made of these stack operations and variations of the above operations (swapn, drop2, etc.) also use a combination of core stack operations.

Syntactic Sugar Stack Operations

- Extend: Expand each element in a stack that is a stack item into separate stack items.



- Compress: Take certain stack items and compress into a single stack item using swap and insert.
- Swap2: Swaps the first item on the stack with the third item.
- Overn: Copies the nth item on the stack to the front.
- Swapn: Swaps the first item on the stack with the nth item.
- Dup2: Duplicates the top two items on the stack.
- Over2: Copies the third item on the stack and places it at the top of the stack.
- Drop2: Drops the top two items from the stack using drop.

These items were considered important and useful to language users, yet were not added to the core because of their ability to be implemented using the existing set of core features. Each of the stack operations that use syntactic sugar also uses at least one core stack operation, adding validity to the selection of core stack operations and reaffirming the decision to remove these features from the core.

Tuples and Operations

Details:

- Tuples containing different types
- Nested Tuples
- Unpacking
- GetFirst: Gets the first item in the tuple
- GetSecond: Gets the second item in the tuple
- SetFirst: Sets the first item in the tuple
- SetSecond: Sets the second item in the tuple
- Tuplelib: Exposes the set of tuple functions

Tuples and the tuple operations are exposed to the user at the library level to abstract away the direct interaction with the underlying array data type. The tuple data type allows for different types to be contained and can have nested tuples. A tuple is created as an inner stack with exactly two items. These items can be retrieved or set using the getfirst, getsecond, setfirst, and setsecond functions. Tuples can also be unpacked using the extend operation that was made for stacks, making them useful for passing variables and results between functions.

Array data type and operations

Details:

- Get: Gets an array element
- Set: Sets an array element
- Concatenation: Concatenates two arrays
- Length: Gets the length of an array
- Range: Creates a new array from the range of another
- Shift Left: Shifts the elements in an array to the left by 1
- Shift Right: Shifts the elements in an array to the right by 1
- Shift Left n: Shifts array elements to the left by n
- Shift Right n: Shifts array elements to the right by n
- Arraylib: Exposes the set of array operations to the user

The array data type is implemented at the library level and provides a set of basic array operations including random access. The array functions are placed into the array library for easy access by the user.

Strings and operations

Details:

- Concatenation: Concatenate two strings together
- Length: Get the number of characters in a string
- Get: Gets a string element
- Set: Sets a string element
- Shift Left: Shifts the elements in a string to the left by 1
- Shift Right: Shifts the elements in a string to the right by 1
- Shift Left n: Shifts string elements to the left by n
- Shift Right n: Shifts string elements to the right by n

The string data type uses the array data type and is implemented at the library level. In this way, the user only needs to interact with the library functions instead of interacting with the stack data type directly. All of the string operations are the same as the array operations, as strings are implemented as arrays of integers where each number represents an ASCII character. An additional feature is added at the core/sugar levels to implement a "newString" function which makes it easier to create strings by creating them directly from ASCII characters instead of from the numerical representation that is used within the language.

Other Features

Details:

- Block
    - Block is used to convert a program into a single executable command. Block is used to build several features throughout the program.
- stdLib
    - Standard library that exposes the tuple library and the array library to the user.

What *level* does each feature fall under (core, sugar, or library), and how did you determine this?

Feature	Level	Reasoning
Boolean Values	Sugar	Can be implemented via Integer of 0 or 1
Integer/Float Values	Core	Essential for math operations and looping.
While Loop	Core	Used for creating for loops and foreach loops.
Swap2, Overn, Dup2, Over2, Drop2, Extend, Compress Stack Operations	Sugar	Extra features for stack operations that are beneficial and can be constructed from the core stack operations.
Insert, Extract, Swap, Dup, Over, Drop Stack Operations	Core	Minimum needed stack operations to implement the extra stack features.
Less Than, Equal, Multiply, Add	Core	These are needed to implement

Operations		the other logical operation features.
Greater Than, And, Or, Negation, Not Operations	Sugar	Can be implemented using the core logical operations and Integers, can be removed from core
AddOne and SubtractOne Operations	Sugar	Useful for looping and not necessary at the core.
If-Then-Else	Core	Needed at the core to implement stack operations and logical operations
For and Foreach Loop	Sugar	Can use the while loop to implement and therefore not needed at the core.
Strings and Operations	Library	Library functions can be exposed to the user while removing the need for the user to interact directly with the array data type. The abstraction augments the user experience.
Tuples and Operations	Library	Library functions can be exposed to the user while removing the need for the user to interact directly with the array data type. The abstraction augments the user experience.
Array Data Types and Operations	Library	Library functions can be exposed to the user while removing the need for the user to interact directly with the stack. The abstraction augments the user experience.

What are the *safety properties* of your language? If you implemented a static type system, describe it here. Otherwise, describe what kinds of errors can occur in your language and how you handle them.

i. Stack underflow can occur when an operation requiring arguments is performed on the stack but the stack is empty or has too few arguments. If this happens, the program will return the error value “Nothing” instead of a stack.
ii. Operations on the incorrect types -- for instance using a stored function for addition. This will cause the program to return the error value “Nothing.”
iii. A function that does not exist can be called, and this will cause the program to return the error value “Nothing.”
iv. If the program inside a While loop never puts a value that evaluates to false at the top of the stack, the program will loop forever. The same behavior can occur if a negative value is used when making for loops or arrays -- both of these expect positive values so a negative one will cause an error that can result in an infinite loop.

3. **Implementation**
   - What *semantic domains* did you choose for your language? How did you decide on these?

     There are several semantic domains chosen for the language. The first is the domain that maps a list of stack items to the Stack type. The StackItem type maps a Number, Stack or Function to a StackItem. The second domain is the Env type that maps a Function type to Maybe Prog. The last semantic domain is one that maps a tuple of a Stack and the Environment to a Maybe type containing a stack and an environment which is then mapped to the Domain type.

   - Are there any unique/interesting aspects of your implementation you’d like to describe?

     The implementation of the language allows for stacks to be placed into another stack. The basis for several features in the language relies on this ability.