CS381 Project Report

Group Members:

- Eric Prather
- Ley Aldinger
- Caleb Ethan Shultz
- Zachary Ian Lee

Git Repository | Project Description | Design Document Requirements | Rubric

Note: Git Repository should be checked out to the branch "run_stuff" if the configuration used in the packaged ".zip" file is not used.

Readme (Plain Text)

```
# README

Names: Eric Prather, Ley Aldinger, Caleb Ethan Shultz, and Zachary Ian Lee
ONID Usernames: prathere, aldingeh, shultzca, leeza

The name of the language is StackLanguage. As its name implies, it is a stack based language
where the stack is a linked list rather than a block of contiguous memory. This feature is
notable since it features strings as a primitive data type and natively supports basic file I/O.
Our language implements only features which make intuitive sense to users and encourages logical
use via descriptive error handling.

# How to execute programs:

## From GHCi

Open the StackLanguage.hs and associated module in GHCi.
Use the command:
        `[loaded_program] = loadFile [path_to_script_file]`
to load a correctly formatted StackLanguage program. To
execute the program, use the command:
        `run [loaded_proram]`

## From command line

A method for running StackLanguage programs directly
from the command line has not yet been determined.

# Example program expected outputs

Example program outputs have not yet been determined.
```

Introduction

The name of the language is StackLanguage. As its name implies, it is a **stack based language** where the stack is a doubly-linked list rather than a block of contiguous memory. This is possible since the actual memory management is done by Haskell, so lower level memory

management functions can be omitted from the implementation. This also loosely enforces good use of the “stack” data structure, as it is impossible to use random access to get to different positions on the stack. It is still possible to move relatively to the “top” of the stack, although this is not a core design principle of the language.

One important design decision in this program was whether or not to put both **instructions** and **values** on the stack, or whether to keep instructions separate. On one hand, keeping instructions on the stack would represent a closer-to-reality paradigm, but because the software is being implemented through Haskell, this imparts no performance benefit. We eventually decided to evaluate instructions through the stack to increase language use learnability for users familiar with assembly-based paradigms.

Design

Language Features

This language supports a modest set of functions at the core level, with extensibility in mind. The syntactic sugar of the language is primarily an alias for complex permutations of core commands which would be commonly used for basic operations in more intuitive imperative languages. Finally, the standard library handles some complex I/O functions, such as inputting specially formatted data onto the stack in sets of preconfigured ways (rather than just inputting files as strings).

Typing is **dynamic**. Functions are defined at runtime, and their contents (including instructions!) can vary over the course of the programs execution.

Core level

A program is a series of single-word commands. A command can be either a **primitive** (Bool, Int, or String) or an **expression**. Primitives are almost always interchangeable (See Primitive Data Types).
At the core level, this language is **turing complete**. A proof of this is provided in another section.

Sugar

The program supports a variety of syntax that takes human readable commands (for example, “get the length of a string”) and translates it to the obtuse arithmetic instruction set. Syntax can also be used to facilitate function definition.

One important syntactic form specifically *not* supported by our language is the resolution of the order of operations. Notably, this language does **not support PEMDAS**. This is because the

arithmetic operations are heavily overloaded in order to conform to the principle of minimality, meaning that they don't always have the exact behaviors that would be expected in standard mathematical calculations. This also helps to make type conversion **unambiguous**.
The only branching in this program is supported through a primitive "branch" command. For more complex.
There are two key flow control instructions in this language which roughly mean "do not execute this immediately" and "do not pop this from the stack after executing". They **always** come directly before another instruction, because they can apply to themselves recursively. This would be an extremely rare use case though, so it's important that these instructions can be quickly appended to other in the language. Thus, we define the following suffixes:

- & - Do not execute the preceding command immediately
- ; - Do not pop the preceding command after executing.

Standard Library

A standard library of functions is included with the main distribution of the language. This includes the following functions:

Safety and Errors

Failing Expressions

Sometimes, an expression requires a certain set of primitives to be on the stack. If there is a type error, a typecast will be attempted. There is a typecast defined for all three types of primitives in this language.

If there are insufficient items on the stack for the operation, the program encounters an error. Instead of performing the operation desired, the string "ERROR: [Command Name]" is written to the stack. This allows for an easy try-catch system and allows the program to continue running after errors as if nothing ever happened. Because this error string can be typecast to whatever subsequent commands require, this language will continue cascading the error until there are no more instructions to execute. This encourages developers to try extra hard to write bug-free code because it dramatically increases the consequences of errors and makes troubleshooting more difficult.

Primitive Data Types

This language handles the three types of primitive as a single super-type. Arithmetic changes its meaning based on specifically which type of primitive was passed into them. Taking inspiration from Javascript, we defined our type-flow arbitrarily and inconsistently. This helps with the **Minimality** of the language, so that developers have a solid chance at getting their desired outcome from any given inter-type operation without having to specify exactly how they want

their type-casting to play out. This inconsistency is also good because it keeps developers on their toes and makes them consider the behavioral definitions very closely, as if they were developing a Javascript application. This means that errors will **not be thrown** for basic operations interchanging primitive types. Specifically:

Addition

OperandA/OperandB	String	Int	Bool
String	Concatenation	(Int->String) Concatenation	(Bool->String) Concatenation
Int	(String->Int) + Int; if string is not int, get length of string	Int + Int	Int + [0 or 1]
Bool	Logical OR (String == “” ? true : false)	Logical OR (Int == 0 ? true : false)	Logical OR

Note that in this language, addition is not necessarily communicative.

Subtraction

OperandA/OperandB	String	Int	Bool
String	Removes all occurrences of the right operand from the left operand.	Remove last [INT] characters from String. Stops at “”	Removes all contiguous [whitespace / non-whitespace] characters from the end of the string
Int	Subtract length of string from int	Integer subtraction	Int - [0 or 1]
Bool	This type of arithmetic string operation is not supported in our language as it is not likely to be representative of the average user’s understanding of subtraction.	Logical AND (Int == 0 ? true : false)	Logical AND

Note that in this language, subtraction is not necessarily the opposite of addition.

Multiplication

OperandA/OperandB	String	Int	Bool
String	Intersperse characters	Repeat string int times	Keep all [Whitespace / Non-whitespace] characters
Int	(String->Int) * Int; if string is not int, get length of string	Int * Int	Int * [0 or 1]
Bool	This type of arithmetic string operation is not supported in our language as it is not likely to be representative of the average user's understanding of multiplication.	XOR	XOR

It is worth noting that we were unable to completely integrate some of these sets of operations. In particular, we ran into trouble while trying to use strings as our Operator A in our subtraction and multiplication functions. We decided that, since those uses are also the most counterintuitive, it would likely be a better design decision to not include them at all than to try to make a fix with no guaranteed robustness or long-term usability.

Proof of Turing Completeness

This language can be proved to be turing complete by demonstrating that it can simulate a turing machine. In this proof, we assume that it has already been proven that a one-ended tape turing machine has been proven to be able to simulate a turing machine. This is achieved by shifting the dataset N to the right with a discrete sub-automata appended whenever the simulation would select an entity over the end of the tape by N spaces.
Then, it must be proven that our language can implement a single-ended turing machine.

First, assume a baseline execution environment with unbounded **active** and **passive** memory. This programming language interpreter initializes a stack at the first available **passive** address, and for each instruction or value passed into it via the programming language input file, it pushes a new item to the stack. The passive memory represents the memory between the bounded end of the tape and the turing machine pointer. The active memory represents the turing machine pointer to the unbounded end of the tape.

In order to be a turing machine, this language must be able to move the turing pointer left by one, right by one, evaluate conditionals, and change the value pointed to. These actions can be represented by the following instructions:

Turing Machine Action	Language
Move towards bounded end	HOLD POP
Move towards unbounded end	RELEASE (if end not reached) -- [new item] (if end reached, append)
Evaluate condition	[Operand1: Tape value] [Operand2: Program constant] [Instruction to execute if equal]& ==
Write	POP PUSH

Observe the way in which active memory is used as a second tape. This is ill-advised in this language, as it is designed so as to be most efficient on operations in passive memory (and uses active memory mainly as a temporary buffer for arithmetic and other operations, rarely exposed to the programmer).

Notably, an additional requirement of this assignment is that this language is able to store functions. **Because this language is turing-complete, the presence of functions is implicit**. However, a separate “function” system is defined for ease of use and grading at the syntactic level.

For more information on how this implementation is attained, see Example Programs.

Implementation

This language is not case sensitive for instructions

Instruction Set (Core)

Flow Control

- HOLD: Consider the current pointed-to value to be active

- RELEASE: Place the active memory cache onto the stack
- LBL: Adds a name to the **static global scope**, permitting global random access
- TP: Defines a contiguous set of primitives or instructions as a type in the static global scope. Types declared via TP are dynamic, not static.
- END: Terminator for LBL and TP.
- INS: insert the value referred to by a name in the static global scope- i.e. a function.
- INV: Swaps the order of the items in active memory

It is an error to have LBL and TP executing at the same time.

Intrinsic IO

- FileIn: Parses the previous file on the stack
- FileOut: Writes the item two items ago to the filesystem using the filename at the previous item.
- Echo: Writes the previous item on the stack to standard output.

Arithmetic & Primitives

- SUM_ACTIVE: Performs arithmetic addition among active primitives
- DIFF_ACTIVE: Performs subtraction among active primitives
- MUL_ACTIVE: Performs multiplication among active primitives
- False: False
- True: True
- [0-9]: Integer value
- EQU: Test all primitives in active memory for equivalence.
- *: All other inputs are considered strings

Instruction Set (Aliases / Macros / Syntax)

- ADD: HOLD HOLD SUM_ACTIVE RELEASE
- SUB: HOLD HOLD DIFF_ACTIVE RELEASE
- MUL: HOLD HOLD MUL_ACTIVE RELEASE
- [*]*&: If command, do not execute immediately.
- +: ADD
- ==: EQU

Basic Stack Manipulation

The inclusion of these instructions are inspired by froth.com.

- Dup: Duplicates the top item on the stack
- Over: Duplicates the item two items back and puts it on top

- Rot: Rotates the previous three items right
- Drop: Drops the top item of the stack

Semantic Domain

Our language works by parsing a list of words into a Program, which is itself a list of Expressions and Primitive values. As such, the semantic domain of our language is determined by our custom “Primitive” data type, which is represented by an Int, Bool, Or String. Every operation will result in one of these data types as an output alongside a stack composed of Primitives and Expressions. We chose to structure our language this way as it allows us to take a minimalist approach to data types while still allowing us to be Turing Complete. We chose to prioritize minimality because we want our language to be approachable for newcomers despite being stack-based and therefore one of the less commonly used language paradigms and felt that using Ints, Bools, and Strings would allow for the fastest adaptation of the language. Our Expressions are similarly simple in concept with addition, subtraction, multiplication, file input and output, if statements, and various common stack manipulations being supported.

Unique and Interesting Aspects

As you may have noticed, our language does not support floating point numbers, characters, or arrays, which many commonly used languages support natively. Despite this, it is still possible to solve most, if not all, problems that would normally require these features. In essence, a floating point number can be represented as an integer with an associated power of ten, a character can be represented as a 1-long string, and an array can be represented as its own stack composed solely of primitives, though random access isn’t permitted. In addition, we support Bools and Strings natively which are less likely to be natively supported (see C and Pascal for examples). In effect, we natively support certain commonly derived types as primitives while requiring that certain common primitives be derived instead.

Example Programs

Details on how to run these example programs are provided in the readme file.

Good Examples

Program 0: “Hello World”

```
1
2
==
12
BranchIfTrue
"Hello world! Math result: "
cur
17
+
+
echo
"hello_world.txt"
fileOut
True
2
BranchIfTrue
"This line will never print to the screen"
echo
"End of program"
echo
```

Program 1:

1

```
1
==
4
BranchIfTrue
10
10
-
```

```
echo
“Everything before this doesn’t exist”
Echo
```

Bad Examples

Example 1: Greeting Message

```
echo
+
“Hello! Running StackLang. Position: ”
Cur
```

This is a bad example because it is written in prefix notation, not postfix notation. The program will read the program from top to bottom, putting things on the stack as it sees them and executing them immediately. Because they are executed immediately and the arguments have not yet been supplied, the program will crash.

Example 2: [no hello] World

```
Hello world!
Echo
```

This language is whitespace delimited and ignores newline characters. Because “Hello” and “World” are not instructions, ints, or booleans, the parser will interpret each as two separate strings. Therefore, the output of this program will be “world!”, not “Hello world!”

Example 2: Equality

```
1
==
1
2
BranchIfTrue
```

This example fails because the equality checking structure is not handled correctly. The == command will test equality of two previously pushed primitive values. Because commands are executed as they arise, attempting to determine equality in this imperative-like fashion will either throw an error or produce unexpected results by checking against erroneous previously pushed values.

Example 3: Disallowed String Structure

```
“-Dont Do This”
echo
```

The way that our language functions currently, this type of command is disallowed. Because the parser is designed to handle the manipulation of positive and negative integers from a list of words, beginning a string with a ‘-’ is disallowed in this language, and doing so will cause an error.

Code

To view the milestone code, see: https://github.com/ValdemarTD/CS381FinalProject