Programming Language Concepts
Language Manual
"No Detriment Policy has not been upheld"lang

Team Member - University Username

Nawab Mir – nm2g18@soton.ac.uk
Dylan Maguire – dm4g17@soton.ac.uk

Table of Contents:

How to use “No Detriment Policy has not been upheld”lang:

Once everything is compiled, there are two methods of execution that come along with “No Detriment Policy has not been upheld”lang. The first method of execution is by executing a program with the interpreter such as “./myinterpreter myProgram.spl”. This will set the user into an environment where they can input from the command line, this input will then be run through an executed program before a result is output. The second method of program execution is by piping data into a program as such, “./myinterpreter myProgram.spl < inputData.txt”.

Much like how ECS piped semester 1 marks into the average for the year:

“./ecsNoDetriment yearAverage.spl < semester1.txt”.

Execution Model

The interpreter for the program uses an adapted CEK-evaluator with *State* being a 4-tuple.

An *Expression* is an operator paired with its operands, and represents the expression that needs to be evaluated to continue program flow.

The *Environment* is a list of 2-tuples that track the stored variables by the user - there is only one global scope within “No Detriment Policy has not been upheld”lang.

Frames track instructions in the case that the operand *Expression*(s) is/are not fully evaluated - this evaluation is necessary to continue with program execution.

The *Kontinuation* is a stack of frames that tracks program execution.

Streams are a list of lists representing the stored (infinite) input matrix.

The transition function *step* transforms the *State* into a new *State* with the IO Monad applied to support stdin and stdout side effects. This transition function may add new tuples to the *Environment*, add a new *Frame* to the *Kontinuation* and/or manipulate the *Streams*. The transition function is applied until the state reaches a terminal state where the final and only frame in the *Kontinuation* is *Done*. Then the *Expression* of the final *State* is the result of the program.

$$Frames : Done \mid IfK\ Expression\ Expression \mid ...$$
$$Expression : EInt\ Int \mid And\ Expression\ Expression \mid ...$$
$$Environment : [(String, Expression)]$$
$$Kontinuation : [Frames]$$
$$Streams : [[Int]]$$

$$State : (Expression, Environment, Kontinuation, Streams)$$
$$step : State \rightarrow IO\ State$$

Syntax

Primitives:

Booleans: ‘True’ and ‘False’
Integers supported up to $2^{63} - 1$

Both integers and booleans are valid programs. That is to say an spl file containing only ‘5’ or ‘True’ would be accepted by the interpreter.

Int operations:

Add (a+a), Minus (a-a), Multiply (a*a), Divide (a/a), Exponential (a^a), Negate (-a)
Equality (a == a), Lesser (a < a), Greater (a > a), Lesser Equal (a <= a), Greater Equal (a >= a)

There is no concept of truthy and falsy variables in this language and thus integers will not behave like booleans. Integers can be stored in variables, streams, and output to stdout. Multiple operators can be applied to integers in one statement and follow standard arithmetic laws, and is not strictly between two integers e.g. 5 + 6 * 2 / 4 would evaluate to 8.

Boolean operations:

Equality (a == a), Logical AND (a && a), Logical OR (a | b), Logical NOT (!a)

Booleans can be stored in variables and output to stdout. Multiple operators can be applied to booleans in one statement and follow boolean algebra laws, and is not strictly between two booleans e.g. True && True | False would evaluate to True.

Variable operations:

Assignment (var = a), Variable Lookup (var)

Variables can be stored as primitives. Variable assignment, or declaration, is required before any manipulation can occur. The left operand can be any alphanumeric combination, and the right, any primitive or primitive operations which evaluate to a single primitive. Variables are mutable and can be looked up within the program environment as a substitute to primitives.

Overriding infix precedence:

(Statement)

Overrides the precedence of infix operators and forces evaluation in the order specified.

Statements:

If:

If (Boolean) Then (Statement) Else (Statement)

If the 'Else' branch is not required, then Statement can simply be replaced without any side-effects by any statement which does not affect the environment like 'False'

While:

While (Boolean) Then (Statement)

This 'While' loop will check if the aforementioned boolean evaluates to be 'True' before continuing on to execute any statements passed afterwards. Thanks to the setup of our 'While' loops it is possible to include an expression and force it to evaluate to either 'True' or 'False'

Example stdIn:	**Example program:**	**Example stdOut:**

1 2 3
4 5 6
7 8 9
	a = 1;
	While(!(a >= 2)) Then (
	  Print(a)
	)
		1
		1
		1
		.
		.
		.

Statement Chaining:

statement1; statement2

This ';' operator causes the operand statements to be executed sequentially, in the order statement 1 -> statement 2 and can be chained any amount of times. Multiple chained statements are recognised as one statement by the parser can be passed to If and While statements.

Output:

Print:

Print (Primitive)

This function is used to print primitives to stdout.

Example stdIn:	**Example program:**	**Example stdOut:**
1 2 3 4 5 6 7 8 9	customVariable = getStream(0, 0); Print(3); Print(True); Print(customVariable)	3 True 1

Stream operations:

incrementStream(n)

Retrieves *n* lines from stdin and stores each stream input into their respective arrays.
The arrays can only be retrieved from and manipulated by built-in stream functions.
Empty lines do not update the arrays.

Example stdin:	**Example program:**	**Resultant stream arrays:**
1 2 3	incrementStream(3)	[1,4,7]
4 5 6		[2,5,8]
7 8 9		[3,6,9]

---

getStream(n, k)

Retrieves the *k*th integer from the *n*th stream array.
n and *k* indexes begin at 0.

Example stream arrays:	**Example program snippet:**	**Snippet evaluation:**
[1,4,7]	getStream(3, 2)	6
[2,5,8]		
[3,6,9]		

---

streamLength(n)

Retrieves the length of the *n*th stream array.
The length of an empty stream is 0.
n and *k* indexes begin at 0.

Example stream arrays:	**Example program snippet:**	**Snippet evaluation:**
[]	streamLength(0)	0
[2,5]	streamLength(1)	3
[3,6,9]	streamLength(2)	2

---

reduceStream(n)

Removes the first index of the *n*th stream array.
Reducing an empty stream array has no effect.

Example stream arrays:	**Example program snippet:**	**Resultant stream arrays:**
[]	reduceStream(0)	[]
[2,5]	reduceStream(2)	[2,5]
[3,6,9]		[6,9]

Additional Features

There are some notable additional features which come included in “No Detriment Policy has not been upheld”lang such as “Error Messages”, “Commenting”.

Stream access:

As the domain of the problem specifies, the language must be able to compute on possibly infinite streams of data. “No Detriment Policy has not been upheld”lang does not store the input file into a matrix and operate on those stream arrays, but rather retrieves directly from stdin when the user declares in their program. This way the language handles potentially infinite streams of data.

Error Messages:

We have some simple error messages including ‘Variable Not Found’ when a variable is trying to be looked up (Appendix 2). The parser also throws errors if the input program does not fit the specified grammar. In such a case the line:column is printed to stdout.

Commenting:

Our language facilitates script commenting and can be added through the form ‘--Comment’. These comments will follow the same format of Haskell's commenting function and can be placed on their own line or on a line with some script function however must be placed after any code. (Appendix 3)

Appendices:

Appendix 1: Programs

1.1

Problem 1: - Double Speed Shuffle

```
incrementStream(2);
While (streamLength(0) >= 2) Then (
    Print(getStream(0, 0));
    Print(getStream(0, 1));
    Print(getStream(1, 0));
    reduceStream(0);
    reduceStream(0);
    reduceStream(1);
    incrementStream(2));

If (streamLength(0) == 1) Then Print(getStream(0,0)) Else False
```

1.2

Problem 2: - Shuffling and Arithmetic

```
incrementStream(1);
While(!(streamLength(0) == 0)) Then (
    a = getStream(0, 0);
    b = getStream(1, 0);
    c = getStream(2, 0);
    Print(c);
    Print(b);
    Print(a);
    Print(a + b);
    Print(b + c);
    reduceStream(0);
    reduceStream(1);
    reduceStream(2);
    incrementStream(1))
```

1.3

Problem 3: - Skip and Prefix

```
Print(0);
incrementStream(2);
While(streamLength(1) == 2) Then(
    Print(getStream(1,1));
    Print(getStream(0,0));
    reduceStream(1);
    reduceStream(1);
    reduceStream(0);
    incrementStream(2)
    );

If (streamLength(1) == 1) Then Print(getStream(0,0)) Else False
```

1.4

Problem 4: - More Stream Arithmetic and Local Reverse

```
incrementStream(3);
While(streamLength(0) >= 3) Then (
    var1 = getStream(0, 2);
    var2 = 2 * getStream(0, 1);
    var3 = (3 * getStream(0, 0)) - 1;
    Print(var1);
    Print(var2);
    Print(var3);
    reduceStream(0);
    reduceStream(0);
    reduceStream(0);
    incrementStream(3))
```

1.5

Problem 5: - Accumulator

```
incrementStream(1);
currentSum = 0;
While (streamLength(0) == 1) Then (
    currentSum = currentSum + getStream(0, 0);
    Print(currentSum);
    reduceStream(0);
    incrementStream(1))
```

1.6

Problem 6: - Two-Three Shuffle

```
incrementStream(3);
While (streamLength(0) >= 0) Then
(
If ((streamLength(0) >= 2) && (streamLength(1) >= 3)) Then
(
    Print(getStream(0, 0));
    Print(getStream(0, 1));
    reduceStream(0);
    reduceStream(0);
            Print(getStream(1, 0));
    Print(getStream(1, 1));
    Print(getStream(1, 2));
    reduceStream(1);
    reduceStream(1);
    reduceStream(1)
)
Else If ((streamLength(0) == 2) && (streamLength(1) == 2)) Then
(
    incrementStream(2);
```

```
    Print(getStream(0, 0));
    Print(getStream(0, 1));
    reduceStream(0);
    reduceStream(0);
    Print(getStream(1, 0));
    Print(getStream(1, 1));
    reduceStream(1);
    reduceStream(1)
)
Else
    incrementStream(1);
    Print(getStream(0, 0));
    reduceStream(0);
    incrementStream(1);
    Print(getStream(0, 0));
    reduceStream(0);
    Print(getStream(1, 0));
    reduceStream(1);
            incrementStream(3))
```

1.7

Problem 7: - Skip Two then Three

```
incrementStream(3);
While (streamLength(0) == 3) Then (
   Print(getStream(0,2));
   reduceStream(0);
   reduceStream(0);
   reduceStream(0);
   incrementStream(4);
If (streamLength(0) == 4) Then (
   Print(getStream(0,3));
   reduceStream(0);
   reduceStream(0);
   reduceStream(0);
   reduceStream(0);
   incrementStream(3)) Else False)
```

1.8

Problem 8: - Checksum Differences

```
incrementStream(1);
counter = 0;
checksum = 0;
While(streamLength(0) == 1) Then (
   a = getStream(0, 0);
   b = getStream(1, 0);
   Print(a);
   Print(b);
   reduceStream(0);
   reduceStream(1);
   checksum = checksum + a - b;
   counter = counter + 1;
If (counter == 5) Then (
   Print(checksum);
   counter = 0;
   checksum = 0)
Else (False);
   incrementStream(1))
```

1.9

Problem 9: - Counter Padding

```
incrementStream(1);
counter = 1;
While (!(streamLength(0) == 0)) Then (
   Print(getStream(0,0));
   reduceStream(0);
   Print(counter);
   counter = counter + 1;
   incrementStream(1))
```

1.10

Problem 10: - Fibonacci Sequences

```
done = False;
While (!done) Then (
   incrementStream(1);

   fibSum = 0;
   i = 0;
   While (i < streamLength(0)) Then (
      level = getStream(0,i);

      a = 1;
      b = 1;
      c = 1;

      j = 2;
      While (j < (streamLength(0) - i)) Then (
         c = a + b;
         a = b;
         b = c;
         j = j + 1
                                --test
      );
      fibSum = fibSum + (c * level);
      i = i + 1
   );
   Print(fibSum)
)
```

Appendix 2: Error Messages

```
C:\Users\Test\Desktop\PLC\Domain-Specific-Language\SubmissionOne\Submit>myinterpreter test.spl
myinterpreter: Variable not found
CallStack (from HasCallStack):
  error, called at .\CEKMachine.hs:80:40 in main:CEKMachine
```

Appendix 3: Commenting

```
1  Print(1234); --Comment Test
2  --Comment Test
3  Print(1234)
```

```
C:\Users\Test\Desktop\PLC\Domain-Specific-Language\SubmissionOne\Submit>myinterpreter test.spl
1234
1234
```