Programming Language Concepts Coursework II

The 'Ouroboros' Programming Language

Ashon Subramanian & Christian Folkesson Church
Student IDs - 29339065 & 29414954

Department of Electronics and Computer Science
University of Southampton
United Kingdom
May 9, 2019

1 The Language's Structure

1.1 Syntax

The language consists of a series of statements. Each statement corresponds to some meaningful operation which the program should execute and must end with a `';'`. Formatting is largely left to the programmer, as newlines and white space are all ignored by the language.

1.2 Basic Data Structures and Types

Our language supports a variety of data types and structures which form the backbone for most of the statements available to the programmer. Booleans are expressions which evaluate to either True or False. Numbers are expressions which evaluate to a positive or negative integer. Streams are an ordered collection of Numbers akin to a traditional list, and Stream Blocks are ordered collections of Streams, again, akin to a list of lists. The value of any of these types can be stated explicitly at any valid point in the program. For any of the types, this is done by pairing some explicit base statement with a collection of relevant operators, including none at all. For Booleans, the base statement must be one of the two keywords `'True'` or `'False'`. For Numbers, this must be some integer, prefixed with `'-'` if negative. For Streams it must be a list of Numbers separated by `','` and enclosed within `'[]'`, and similarly for Stream Blocks, it must be a list of Streams separated by `','` and enclosed within `'{}'`.

1.3 Variables

Stream Blocks can be stored and accessed through the use of variables. Variables are declared with the `'var'` keyword, and use the variable assignment operator `'='` to assign the variable to some value. The variable's name can then be used in any context where a Stream Block would be accepted. The variable's name must be entirely alphabetic and uppercase, and distinct from any reserved keywords (see Appendix B).

```
var EXAMPLE = {[0,1,2],[3,4,5]};
```

1.4 Importing Data

We use a mechanism called 'importing' to allow the language to access data from external sources. This involves passing a correctly formatted Stream Block (see Appendix A) into the program's STDIN. The import statement then assigns the data some variable, which can be used normally. This consists of the keywords `'import as'` followed by the variable name.

```
import as DATA;
```

1.5 Printing

A print statement will output the value of some Stream Block on STDOUT once the program has finished execution. A print statement consists of the keyword `'print'` followed by some Stream Block.

```
print {[1,2,3]};
```

1.6 Indexing

A mechanism called 'indexing' can be used to access the values within a Stream Block or Stream. Indexing a Stream Block will result in a Stream, while indexing a Stream will result in a Number. An indexing expression consists of a Stream or Stream Block followed by the indexing operator `'@'` and the position of the desired element, starting at 0.

```
{[1,2,3,4]} @ 2
```

1.7 Length Operations

The length operator '#' is used to determine the length of a Stream Block or Stream. The length of a Stream Block is the number of Streams it contains, meanwhile the length of a Stream is the number of Numbers it contains.

```
#[1,2,3,4]
```

1.8 Looping

Our language natively provides a loop structure to iterate over some collection of statements while a Boolean condition is true. This consists of three components; the `'loop'` keyword followed by a Boolean condition given in parentheses and collection of statements in parentheses following that.

```
var I = {[0]};

loop (I @0 @0 < 5)(
    var I = {[(I @0 @0) +1]};
)
```

The loop statement is the only one which does not end with a `';'`. This is because it is designed to span over many lines. Note that the loop statement can be used to construct 'for' loops and 'if' statements (see subsection 3.3).

1.9 Concatenation

The concatenation operator is an operator that merges two streams together resulting in a new Stream consisting of the entire first Stream, followed by the entire second Stream.

```
[1,2,3] & [4,5,6]
```

1.10 Numerical Operations

Basic mathematical operations (addition, subtraction, and multiplication) can be performed on numerical expressions. Note that multiplication has the highest precedence.

```
1 + 5 - 3 * 5
```

1.11 Boolean Operations

Similarly, basic Boolean operations (`'AND'`, `'NOT'`, `'OR'`) can be performed on any Boolean expression. In addition, equality operations(`'=='`, `'>'`, `'<'`) can be used to reason about Numbers. Note that the equality operator is `'=='` and that `'='` is used for variable declarations.

```
1 + 1 > 3 OR 3 == 3
```

2 Additional Features

2.1 Commenting

The programmer can include comments at any point in the program using `'||'`. Any text following `'||'` will be ignored during the execution of the program.

2.2 Errors

Our language provides rudimentary error messages in the case of an error in the program. In the case of a syntax error, the program will cease execution and provide a `'Parse Error'`. If the program were to try and index a position which does not exists, the program would provide an `'Index too large'` error. In the case that a variable is used which has not been assigned a value, the program will provide a `'Unbound variable'` error and give the name of the unbound variable. Note than many other errors will simply be interpreted as syntax errors.

2.3 Design Patterns

Several design patterns exist in our language which may aid the programmer in creating solutions to problems. The Singleton Element pattern allows storage and access of a single number. It's done by wrapping a number in a Stream and then a Stream block, and accessed by indexing both of these to the first position. Building on this, the Indexed Loop can be used as a 'for loop'. This is done by declaring a variable using the Singleton Element pattern, then creating a loop with a condition involving this value, and increasing the value within the loop. An example for both of these can be seen in subsection 2.8. Finally, an 'if statement' can be created by declaring a variable with some value using the Singleton Element pattern, then creating a loop, where the Boolean condition involves the variable being equal to its initial value and some other expression, and then changing the value of the this within the loop.

```
var IF = {[0]};

loop (IF @0 @0 == 0 AND ... )(
    var IF = {[1]};
)
```

3 Appendix A - Stream Block Format

A Stream Block is an ordered collection of Streams, similarly, a Stream is an ordered collection of Numbers. These are formatted in an intuitive but very specific way, and this must be adhered to when providing Stream Blocks as input to the program. Only a single Stream Block can be passed into the program on STDIN but there is no limitation as to the number of Streams given in a stream block, or the length of the Streams, although the length of all Streams must be equal. Streams are given in the form of a vertical column of numbers, where each number appears on a new line. Additional Streams are added as new columns, separated by a single white space.

```
1 4
2 3      ->      {[1,2,3],[4,3,2]}
3 2
```

In addition, negative numbers can be included by prefixing the number with a '-'. Note that because many of the numbers may not be the same length, the alignment of the columns may shift and give the appearance that the streams are not aligned. This will not be a problem as long as there is a there is a white space between each Number in each line, as the program will ignore the offset caused by the length of each Number.

```
1234 5 89 5
6 42 -6 2      ->    {[1234,6,6],[5,42,9],[89,-6,2],[5,2,4]}
6 9 2 4
```

Each line must end with an End of Line character ('EOL') and the file must end with an End of File marker ('EOF').

4 Appendix B - Reserved Keywords & Symbols

```
print
var
import
as
loop
True
False
AND
OR
NOT
=
==
+
-
*
&
@
#
<
>
```

5 Appendix C - Solutions

The following are our solutions to all ten problems

5.1 Problem 1

```
import as DATA;

var I = {[0]};
var OUT = {[0]};

loop ((I @0 @0) < #(DATA @ 0) - 1)
(
    var OUT = {OUT @0 & [DATA @0 @(I @0 @0)]};
    var I = {[(I @0 @0) +1]};
)

print OUT;
```

5.2 Problem 2

```
import as DATA;
print {DATA @0, DATA @0};
```

5.3 Problem 3

```
import as DATA;

var OUT = {[]};
var I = {[0]};

loop ((I @0 @0) < #(DATA @ 0))
(
    var AVAL = {[DATA @0 @(I @0 @0)]};
    var BVAL = {[DATA @1 @(I @0 @0)]};
    var RESULT = {[(AVAL @0 @0) + 3 * (BVAL @0 @0)]};
    var OUT = {OUT @0 & RESULT @0};
    var I = {[(I @0 @0) +1]};
)

print OUT;
```

5.4 Problem 4

```
import as DATA;

var OUT = {[]};
var I = {[0]};
var ACCUMULATOR = {[0]};

loop ((I @0 @0) < #(DATA @ 0))
(
    var VAL = {[DATA @0 @(I @0 @0)]};
    var ACCUMULATOR = {[ (ACCUMULATOR @0 @0) + (VAL @0 @0) ]};
    var OUT = {OUT @0 & ACCUMULATOR @0};
    var I = {[(I @0 @0) +1]};
)

print OUT;
```

5.5 Problem 5

```
import as DATA;

var FIB = {[1,1]};
var I = {[0]};
var OUT = {[]};

loop (I @0 @0 < #(DATA @0))
(
    var FIB = {[(FIB @0 @0) + (FIB @0 @1)] & FIB @0 };
    var J = {[0]};
    var NEXT = {[0]};

    loop (J @0 @0 < (I @0 @0)+1)
    (
        var PRODUCT = {[ (DATA @0 @(J @0 @0)) * (FIB @0 @((J @0 @0)+2 )) ]};
        var NEXT = {[(NEXT @0 @0) + (PRODUCT @0 @0)]};
        var J = {[(J @0 @0) + 1]};
    )

    var OUT = {OUT @0 & [NEXT @0 @0]};
    var I = {[(I @0 @0) + 1]};
)

print OUT;
```

5.6 Problem 6

```
import as DATA;

var I = {[0]};
var OUT = {[0]};

loop ((I @0 @0) < #(DATA @ 0) - 1)
(
    var OUT = {OUT @0 & [DATA @0 @(I @0 @0)]};
    var I = {[(I @0 @0) +1]};
)

print {DATA @0, OUT @0};
```

5.7 Problem 7

```
import as DATA;

var I = {[0]};
var OUT = {[]};

loop ((I @0 @0) < #(DATA @ 0))
(
    var AVAL = {[DATA @0 @(I @0 @0)]};
    var BVAL = {[DATA @1 @(I @0 @0)]};
    var RESULT = {[(AVAL @0 @0) - (BVAL @0 @0)]};
    var OUT = {OUT @0 & RESULT @0};
    var I = {[(I @0 @0) +1]};
)

print {OUT @0, DATA @0};
```

5.8 Problem 8

```
import as ADATA;

var I = {[0]};
var BDATA = {[0]};
var OUT = {[]};

loop ((I @0 @0) < #(ADATA @0))
(
    var BDATA = {BDATA @0 & [ADATA @0 @(I @0 @0)]};
    var AVAL = {[ADATA @0 @(I @0 @0)]};
    var BVAL = {[BDATA @0 @(I @0 @0)]};
    var RESULT = {[(AVAL @0 @0) + (BVAL @0 @0)]};
    var OUT = {OUT @0 & RESULT @0};
    var I = {[(I @0 @0) +1]};
)

print OUT;
```

5.9 Problem 9

```
import as DATA;

var NAT = {[1]};
var I = {[0]};
var OUT = {[]};

loop (I @0 @0 < #(DATA @0))
(
    var NAT = {[(NAT @0 @0) + 1] & NAT @0};
    var J = {[0]};
    var NEXT = {[0]};

    loop (J @0 @0 < (I @0 @0)+1)
    (
        var PRODUCT = {[ (DATA @0 @(J @0 @0)) * (NAT @0 @(J @0 @0)+1) ]};
        var NEXT = {[(NEXT @0 @0) + (PRODUCT @0 @0)]};
        var J = {[(J @0 @0) + 1]};
    )
    var OUT = {OUT @0 & [NEXT @0 @0]};
    var I = {[(I @0 @0) + 1]};
)

print OUT
```

5.10 Problem 10

```
import as ADATA;

var I = {[0]};
var BDATA = {[0,0]};
var OUT = {[]};

loop ((I @0 @0) < #(ADATA @0))
(
    var AVAL = {[ADATA @0 @(I @0 @0)]};
    var BVAL = {[BDATA @0 @(I @0 @0)]};
    var RESULT = {[(AVAL @0 @0) + (BVAL @0 @0)]};
    var BDATA = {BDATA @0 & RESULT @0};
    var OUT = {OUT @0 & RESULT @0};
    var I = {[(I @0 @0) +1]};
)

print OUT;
```