Programming Language Manual

Streamer

1. Introduction

The aim of our programming language is to receive a stream of integers, compute on it and output a new stream of integers. The syntax of the language is inspired by the syntax of Java.

2. Types

The language provides support for four types of values. However, only **Int**, **Bool** and **Stream** can be used on their own as variables. **Var** type is strictly only for declaring variables and it holds the type.

- 32-bit positive and negative Integer number

```
<Int> ::= <digit> | "-" <digit> | <Int> <digit>
```

- 32-bit Strings

```
<Var> ::= <Int> | <Bool> | <Stream>
```

- Booleans

```
<Bool> ::= true | false
```

- Streams composed of lists of Integer values

```
<Stream> ::= <[[Int]]> | <[Int]>
```

3. Syntax

The language is composed of expressions that are parsed by the compiler. Each expression ends with a semi-colon, unless that is the end of the program, in which case the semi-colon is omitted.

Example 1:

```
\* Simple declaration of variables */;
Int x = 5;
Int y = 10;
\* Already declared variables can be used in declaration of new
variables */;
Bool bool = ((x + y) < 16);
\* Prints the result of bool at the end of the program */;
bool
```

The code snippet above declares two integer values and then prints their sum. The semi-colon at the end of each declaration indicates the end of an expression and the absence of

semi-colon in the last line indicates that this is the last line of the program, so the result of that is printed.

Format of the expressions:

- Base expression format:

```
<expression> ::= <expression> | <expression> ";" <expression>
```

- Variable format:

```
<type> <string> ::= <value>
```

- While Loop format:

```
while "(" <condition> ")" "{" <expression> "}"
```

- If-then-else format:

```
if "(" <condition> ")" then <expression> else <expression>
```

- Comment format:

```
"\*" <string> "*/"
```

    - Comments are also parsed as expressions by the compiler, so at the end of each comment, **there should be a semi-colon**

Example 2:

```
Int x = 5;
Int y = 10;
while (x < y) {
        Int y = y - 1;
        Int x = x + 1
};
x
```

You will notice that there is no semi-colon in the last line of the code in the while loop. The reason for this is simple. The while loop can read a complex expression (expression composed of multiple expressions). The last line has no semi-colon to indicate that this is the last part of the complex expression that is being computed inside of the loop. After the condition of the while is no longer satisfied, the loop breaks, and x is printed.

4. Stream

The stream type is the most complex one in the language. The stream type represents a list of lists of integer values. Additionally, it supports several functions that allow us to operate on the stream. They make the stream the most powerful tool for data manipulation of the language.

Creating a stream

Creating a stream is very intuitive and easy. **It is important to notice that we cannot create an empty stream.** The language doesn't support empty streams, so all newly created streams should have at least one element in them.

```
\* Both streams represent the same stream -> [[0]]
The reason for this is that [0] is automatically
parsed to [[0]] by the compiler */;
Stream stream = [[0]];
Stream stream2 = [0]
```

The language doesn't support declaring new streams with variables. **This means that the code below won't compile**.

```
\* This is not allowed because the stream must be first added
to the environment of the program before the other variables in the
environment can interact with it*/;
Int x = 1;
Stream stream = [[x]]
```

Using the stream

There are a lot of operations that the stream type supports. Check *Example 1 and Example 2* in Appendix 6.2 for additional information.

5. While Loop and If-Then-Else

The language supports while loop and if-then-else statements. They work intuitively and as in many other languages. This section will provide a simple example about the syntax and logic of these features.

Example 1:

```
Int x = 5;
Int y = 10;
while (x < y) {
        if (x == (y - 1))
        then (
                Int x = y * 2
```

```
        ) else (
                Int y = y - 1;
                Int x = x + 1
        )
};
x
```

There is a more complex example that showcases how the language can produce the Fibonacci sequence (*Example 3* in Appendix 6.2).

if-then-else

You will notice that we can include multiple expressions in the **then** and **else** part of the if-then-else statement. This is possible if we surround the expressions with round brackets. **It is very important to notice that the semi-colon at the last line (right before closing the brackets) must be omitted.** The reason for this is that the syntax (section 3) of **if-then-else** doesn't use semi-colons to separate the three key words (if, then and else). This is because if-then-else is counted as a **single expression** and not 3 separate expressions.

while loop

The while loop has a condition that must be satisfied for the program to start the loop. After the condition is no longer satisfied, the program doesn't loop anymore. The condition **must be surrounded by brackets.** After the condition we use square brackets to mark the start and the end of loop. The while loop is a **single expression** that must end with a semi-colon after closing the square brackets. Here the case is the same as in the if-then-else statement, the last line before closing the square brackets **must not end with semi-colon** for the same reason as above (check if-then-else part of this section or the section 3 of the document).

6. Appendix

6.1 Additional Features

6.1.1 Type Checking

The language also supports type checking. This feature makes it easier to troubleshoot any problems that are caused by wrong types of arguments in the expressions.
The type checking recognizes three types: **TyStream**, **TyBool**, **TyInt**

Example 1:

```
5 + true
Print to stderr: Type Mismatch: TyBool is not TyInt
```

Example 2:

```
Int x = 0; Bool y = true; x == y
Print to stderr: Type Mismatch: TyBool is not TyInt
```

Example 3:

```
Stream stream = [[1]]; stream + 1
Print to stderr: Type Mismatch: TyInt is not TyStream
```

Example 4:

```
if 5 then 1 else 0
Print to stderr: Type Mismatch: TyInt is not TyBool
```

Example 5:

```
Stream stream = [[1]]; stream.add(true)
Print to stderr: Type Mismatch: TyBool is not TyStream
```

Example 6:

```
Stream stream = [[1]]; true + stream.size()
Print to stderr: Type Mismatch: TyBool is not TyInt
```

Example 7:

```
Stream stream = [[1]]; 2 + stream.add([2])
Print to stderr: Type Mismatch: TyStream is not TyInt
```

6.1.2 Error Reporting

The language supports simple error reporting that highlights any lexical, parsing or binding errors found in the code.

Example 1:

```
Int x = y; x
Print to stderr: Variable binding not found
```

Example 2:

```
Int x = 10; x ? 2
Print to stderr: lexical error at line 1, column 15
```

Example 3:

```
Int x = ; x
Print to stderr: Parse error at line:column 1:9
```

6.1.3 Comments

Comments are also supported by Streamer. They are parsed as expressions by the compiler, so they also must end with a semi colon.

Example 1:

```
\* This is a comment. */;
Int x = 10; x
Output: 10
```

6.2 Code Examples

Example 1 – Stream Operations

```
Stream -> Stream
stream.add([<value>])-> Adds new list with <value> to the end of
the stream. Value can be a stream or an Int value.

Stream -> Int
stream.size() -> Returns the size of stream

Stream -> Stream
stream.get(<Int>) -> Returns the <Int>-th list from the stream
```

```
Stream -> Int
stream.getElem(<Int1>, <Int2>) -> Returns the <Int1>-th elem from
the <Int2>-th list in the stream

Stream -> Stream
stream.remove(<Int>) -> Dequeues the first <Int> lists from the
stream

Stream -> Stream
stream.delete(<Int>) -> Deletes the <Int>-th list in the stream

Stream -> Stream
stream.zip(<Stream>) -> Zips the lists of the two streams
E.g. [[1],[2],[3]].zip([[0],[1]]) -> [[1,0],[2,1],[3]]
```

Example 2 – Examples of Stream Operations

```
Int x = 5;
Stream stream = [[0]];

\* Adds x to the stream */;
stream.add([x]); \* Result: [[0],[5]] */;

\* Adds 0 to the stream */;
stream.add([0]); \* Result: [[0],[5],[0]] */;

\* Dequeue 1 elem from the beginning of the stream */;
stream.remove(1); \* Result: [[5],[0]] */;

\* Deletes the n-th element in the stream */;
stream.delete(0); \* Result: [[0]] */;

\* Zips the elems of two streams into one */;
stream.zip(stream); \* Result: zip [[0]] with [[0]] -> [[0,0]] */;

\* Returns an integer value with the size of the stream */;
stream.size();

\* Returns the first element in the first stream */;
stream.get(0,0); \* Result: 0 */;

\* Stream elements can be used as arguments for calculations */;
```

```
\* Expression: 5 * 0 + 1 -> Prints 1 at the end of the program  */;
x * stream.get(0,0) + stream.size()
```

Example 3 – Fibonacci Sequence

```
\* Variable keeps track of lines in the file */;
Int lines = 0;
\* Variable keeps track of the sum of elems in the sequence */;
Int sum = 0;
\* New stream that will hold the Fibonacci sequence */;
Stream newStream = [[0]];
\* While loop is active until we reach the end of the input file
(or the last list in the stream) */;
while (lines < stream.size()) {
        \* Base case 1. Add first elem of input stream */;
        if (lines == 0)
        then (
                Int sum = stream.getElem(0,lines);
                newStream.add([sum]);
                Int lines = (lines + 1)
        )
        else (
                \* Base case 2. Sum first elem of the input stream
                With the first entry in the stream that holds the
                first value of the Fibonacci sequence */;
                if (lines == 1)
                then (
                        Int sum = (stream.getElem(0,lines) +
newStream.getElem(0,(lines)));
                        newStream.add([sum]);
                        Int lines = (lines + 1)
                )

                 else (
                \* This part computes the rest of the Fibonacci
                sequence by summing the last 2 elements from the
                new stream with the next element from the input
                stream */;
                        Int sum = (stream.getElem(0,lines) +
newStream.getElem(0,(lines)) + newStream.getElem(0,(lines - 1)));
                        newStream.add([sum]);
                        Int lines = (lines + 1)
                )
```

```
        )
};
\* Prints the Fibonacci sequence after deleting the first element
which is not part of the Fibonacci sequence */;
newStream.delete(0)
```

6.3 Submitted Programs

Pr1.spl

```
Stream newStream = [[0]];
newStream.add([stream]);
newStream.delete(stream.size())
```

Pr2.spl

```
stream.zip(stream)
```

Pr3.spl

```
Int lines = 0;
Int sum = 0;
Stream newStream = [[0,0]];
while (lines < stream.size()){
        Int sum =
(stream.getElem(0,lines)+(3*stream.getElem(1,lines)));
        newStream.add([sum]);
        Int lines = (lines + 1)
};
newStream.delete(0)
```

Pr4.spl

```
Int lines = 0;
Int sum = 0;
Stream newStream = [[0]];
while (lines < (stream.size())){
        Int sum = (sum + stream.getElem(0,lines));
        newStream.add([sum]);
```

```
        Int lines = (lines +1)
};
newStream.delete(0)
```

Pr5.spl

```
Int lines = 0;
Int sum = 0;
Stream newStream = [[0]];
while (lines < stream.size()) {
        if (lines == 0)
        then (
                Int sum = stream.getElem(0,lines);
                newStream.add([sum]);
                Int lines = (lines + 1)
        )
        else (
                if (lines == 1)
                then (
                        Int sum = (stream.getElem(0,lines) +
newStream.getElem(0,(lines)));
                        newStream.add([sum]);
                        Int lines = (lines + 1)
                )

                 else (
                        Int sum = (stream.getElem(0,lines) +
newStream.getElem(0,(lines)) + newStream.getElem(0,(lines - 1)));
                        newStream.add([sum]);
                        Int lines = (lines + 1)
                )
        )
};
newStream.delete(0)
```

Pr6.spl

```
Stream newStream = [[0]];
newStream.add([stream]);
stream.zip(y);
stream.delete(x.size() - 1)
```

Pr7.spl

```
Stream arit = [[0]];
Stream nums = [[0]];
Int temp = 0;
Int lines = 0;
while (lines < (stream.size())){
        Int temp = stream.getElem(lines, 1);
        nums.add([temp]);
        Int temp = stream.getElem(lines, 0) -
stream.getElem(lines,1);
        arit.add([temp]);
        Int lines = (lines +1)
};
arit.remove(1);
nums.remove(1);
nums.zip(arit)
```

Pr8.spl

```
Stream newStream = [[0]];
Int temp = 0;
Int lines = 0;
while (lines < (stream.size())){
        if(lines == 0)
        then (
                Int temp = 0 + stream.getElem(0,0);
                newStream.add([temp]);
                Int lines = (lines + 1)
        )
        else (
                Int temp1 = lines - 1;
                Int temp = stream.getElem(0, temp1) +
stream.getElem(0, lines);
                newStream.add([temp]);
                Int lines = (lines + 1)
        )
};
newStream.remove(1)
```

Pr9.spl

```
Stream newStream = [[0]];
Int lines = 0;
Int sum = 0;
while (lines < (stream.size())){
        if(lines == 0)
        then (
                Int sum = stream.getElem(0,0);
                newStream.add([sum]);
                Int lines = (lines + 1)
        )
        else (
                Int counter = 0;
                while (counter < lines + 1){
                        Int sum = sum + stream.getElem(0,counter);
                        Int counter = (counter + 1)
                };
                newStream.add([sum]);
                Int lines = (lines + 1)
        )
};
newStream.remove(1)
```

Pr10.spl

```
Stream newStream = [[0]];
Int lines = 0;
Int sum = 0;
Int counter = 0;
while (lines < (stream.size())){
        if(lines == 0)
        then (
                Int sum = stream.getElem(0,0);
                Int sum2 = stream.getElem(0,1);
                newStream.add([sum]);
                newStream.add([sum2]);
                Int lines = (lines + 2)
        )
        else (
                Int sum = newStream.getElem(0,counter) +
stream.getElem(0,lines);
                newStream.add([sum]);
```

```
                Int lines = (lines + 1);
                Int counter = (counter + 1)
        )
};
newStream.remove(1)
```