University of Southampton
School of Electronics and Computer Science

SPL Domain-Specific Language Coursework Report COMP2212

George Garrington, gg2g17, ID: 29335116
Irina Nikolova, in1u18, ID: 30098467

14/05/2020

Introduction

The syntax of our language is inspired mostly by Java with regards to an iterative style, using curly braces for loops and if/else statements, but also by Haskell partly by incorporating the keywords “take” and “drop” with similar functionality and also in the behaviour of the interpreter. We aimed to make our language as clean, readable and minimalistic as possible. All variables in our language have global scope which is actually an unusually useful quality for the given domain; see “Observations”.

Execution Model

Our language has a similar execution model to other C style languages in that each of the semicolon-separated expressions are indivisible and independent units of work which are performed sequentially and quite literally in the order they are written in the program, and our “while”, “for”, “if” and “else” keywords also manipulate the order of expressions as in other C style languages. The interpreter executes the parse tree in a DFS like fashion with “Seq” expressions behaving as the nodes of the tree; it begins by performing the leftmost innermost expression in the parse tree and continues in evaluating the rest of the expressions in the standard order specified by DFS. The interpreter keeps track of the state of the variables and streams in the program by storing this in a “State” data type, manipulating this whenever state is changed and passing this to each new call of the expression evaluation method.

Expression Types

- **Exp:** an expression that “does” something i.e. invokes the interpreter to perform some action
- **BExp**: an expression which evaluates to some boolean value
- **IntExp**: an expression which evaluates to some integer value (except “take”, see “Observations”)

Lexical Analysis

Category	**Syntax**	**Explanation**
Semicolon	e1; e2 e;	Exp expressions are only recognised if they are declared with a semicolon at the end, with “e1; e2” signifying 2 expressions one after the other and “e;” denoting a single expression. Within the grammar, a sequence (Seq e1 e2) of expressions is actually an Exp expression in itself
Exp keywords	print(ix1, ix2, ... , ixn) streams[n].drop(m) streams[n].reverse nothing end	“print” takes a comma-separated list of IntExp’s ix1 ... ixn and prints the integer value of each on a new line. The streams in the input file are referenced in this zero-indexed notation streams[n]. “drop” drops m of the first elements from stream n similar to Haskell, except conversely to Haskell as our language has state this actually changes the state of the stream too. “reverse” reverses stream n. “end” causes execution to terminate and “nothing” does just that - nothing!

Variable declaration/ manipulation	var = ix
	
	syntactic sugars:
	
	var += ix
	var -= ix
	var *= ix
		Our language does not distinguish between variable declaration and manipulation; they are synonymous. “var” here denotes the string name of the variable to be declared/updated and ix denotes an IntExp expression. “var = ix” will add/update the variable name “var” paired with the evaluated int value in the interpreter variable data structure. “+=”, “-=”, “*=” and “/=” as in Java will

	var /= ix var++ var--	perform a binary operation on the values of “var” and “ix” and store the result in var. ++ and -- like in Java will increment/decrement the variable. All these expressions are Exp expressions.
While loop	while(bx) {e}	While the boolean expression bx evaluates to true, the expression “e” within the curly braces will be repeated. The while loop is an Exp expression.
For loop	for(x1, … ,xn; bx; y1, … ,yn) {e}	First, perform arbitrarily many initialisation (Exp type) expressions in the comma-separated list of Exp expressions x1 … xn, then do a loop where bx (a BExp) is the satisfying condition to check before each iteration and y1 … yn is a comma-separated list of incremental (Exp type) expressions to perform on each iteration after performing the (Exp type) expression “e” in curly braces. A for loop is syntactic sugar for this common pattern that can also be performed with a while loop. The for loop is an Exp expression.
Arithmetic Operators	+ (addition) - (subtraction, negation) * (multiplication) / (division) % (modulo) ^ (exponentiation)	These work on IntExp expressions only e.g. for addition the usage is “IntExp + IntExp”. Examples of IntExp expressions can be found below.
IntExp keywords	streams[n].length streams[n].peek streams[n].take maxOf(ix1, ix2) minOf(ix1, ix2)	“length” returns the length of the stream, “peek” returns the value of the head of the stream; “take” also does this however it has a side effect of also removing this from the stream; see “Observations” for more detail. “maxOf” and “minOf” return the largest/smallest IntExp of those given as arguments: ix1 and ix2
Binary Boolean operators	== (boolean equality comparison) != (boolean inequality comparison) && (and) || (or)	Similarly to arithmetic operators working exclusively with IntExp expressions, these also only work with BExp expressions e.g. “BExp && BExp”.
Binary IntExp comparison operators	> (greater than) >= (greater than or equal) < (less than) <= (less than or equal)	The usage is e.g. “IntExp > IntExp” returning true or false depending on if the operator condition was satisfied or not. Note that these operators when comparing operands are actually BExp expressions in themselves
Unary Boolean Operators	! bx	Negates the value of the BExp “bx”
BExp keywords	streams[n].empty true false	“empty” returns true if the stream is empty and false otherwise. “true” and “false” denote the literal boolean values true and false

Observations

Global variable scope actually turned out to be a useful quality for some problems, for instance in our solution to problem 8 we declared a variable within an if statement and then accessed it in another if statement further along in the program; for the given domain we believe this is suitable as only relatively simple manipulation and operations are required to achieve the expected output sequences, and we believe the global scope of all variables should never pose a significant issue for these types of problems.

Although simple, the “end” keyword proved to be useful e.g. in problem 8 when no more of the required output sequence can possibly be printed then execution can simply be terminated in order to avoid wasting computation; we have also made its use convention within our language at the end of a program to both clearly show where the program terminates and also ensure that the program does in fact end (even though it is in theory unnecessary as the interpreter can also finish by running out of expressions to evaluate). We felt the “nothing” keyword was a necessary addition as e.g. one may wish to only perform some expression if a condition is true and do nothing if it is false (in an if/else statement).

Despite lacking typed variables, one benefit of this is that our language actually has an integrated type system that does not require runtime checks and runtime type errors are simply *not possible*. For instance wherever a BExp is expected, if one tried to put “7” (an IntExp) in its place they would receive a parse error and the program would never even run, so one could argue our language in fact has strong, static typing.

“take” is the only “non-pure” IntExp expression which as well as evaluating to an integer value, it also *does* something and has a side effect i.e. removing the head of the stream, so we had to take into account a possible change in state whenever evaluating an IntExp; for this, we took inspiration from the Haskell IO ideology to account for this possible change in state and had the evalInt method (evaluates an IntExp to its int value) return the evaluated int value paired with the possibly changed state in a tuple, where our “State” is a type synonym for a tuple consisting of the list of variables (string and int pairs) and a list of int lists representing the streams from the input file.

Limitations / What we could improve

Our language has no kind of list or data structure which may make some problems impossible to solve, so this is something we would look to implement e.g. maybe a simple list or stack. A type system to check variable types and declaring variables with different types such as list, bool or int would have been a logical extension also.

Extension: Multiline, Inline & Single Line Comments

C style syntax: single line usage is “//” at the start of a line, multi-line/inline usage is “/*” followed by the comment body of text with “*/” at the end

Extension: Error Handling

The interpreter will first calculate the expected number of streams by searching the parse tree for the highest referenced stream index if no stream input file is given and instead run the program with this many empty streams. Due to complications with changes in state, the interpreter does not allow “take” to be used within IntExp expressions in boolean IntExp comparisons and will search the parse tree before running the program to make sure this is satisfied. Other trivial error checks have been implemented and screenshots of all of these can be found in the appendix.

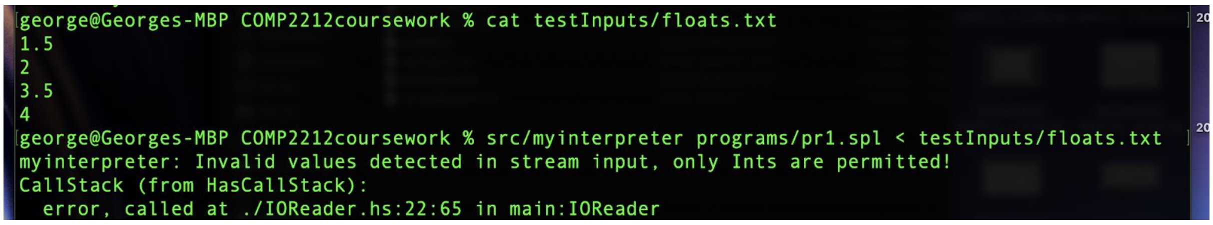
Extension: Syntax highlighting support in Notepad++ text editor

Screenshots of this can be found in the appendix, and the XML syntax file can be found in the handin

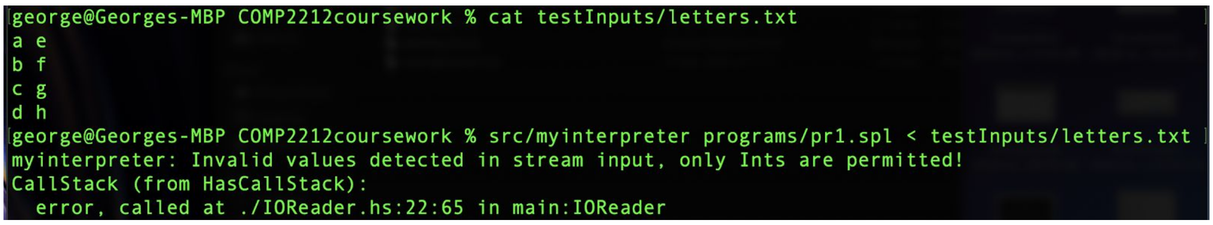
Appendix

Error Handling

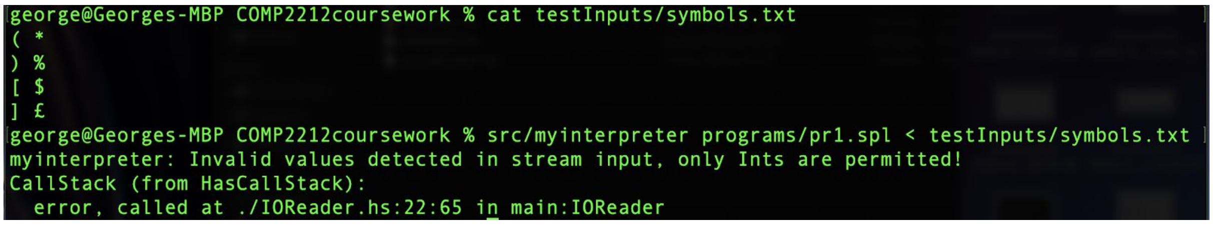
Only allowing integer values in the stream, not allowing floats:


```
george@Georges-MBP COMP2212coursework % cat testInputs/floats.txt
1.5
2
3.5
4
george@Georges-MBP COMP2212coursework % src/myinterpreter programs/pr1.spl < testInputs/floats.txt
myinterpreter: Invalid values detected in stream input, only Ints are permitted!
CallStack (from HasCallStack):
  error, called at ./IOReader.hs:22:65 in main:IOReader
```

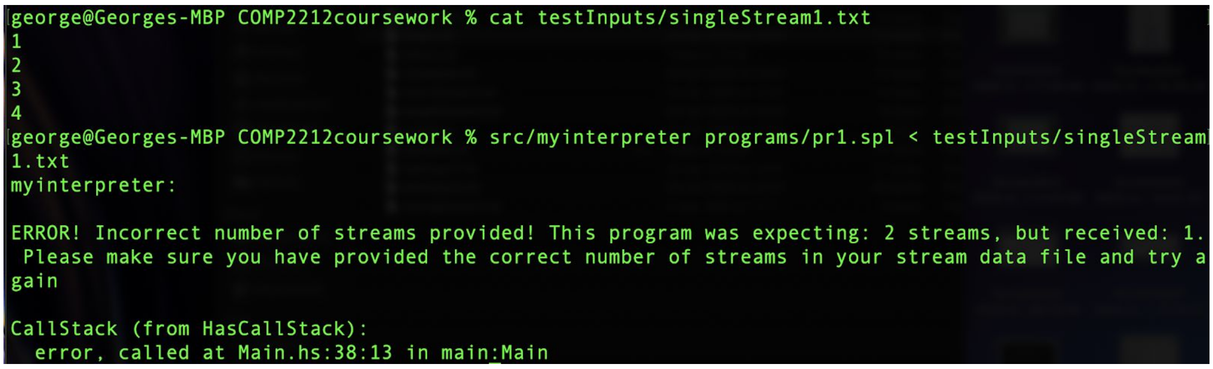
Only allowing integer values in the stream, not allowing chars:


```
george@Georges-MBP COMP2212coursework % cat testInputs/letters.txt
a e
b f
c g
d h
george@Georges-MBP COMP2212coursework % src/myinterpreter programs/pr1.spl < testInputs/letters.txt
myinterpreter: Invalid values detected in stream input, only Ints are permitted!
CallStack (from HasCallStack):
  error, called at ./IOReader.hs:22:65 in main:IOReader
```

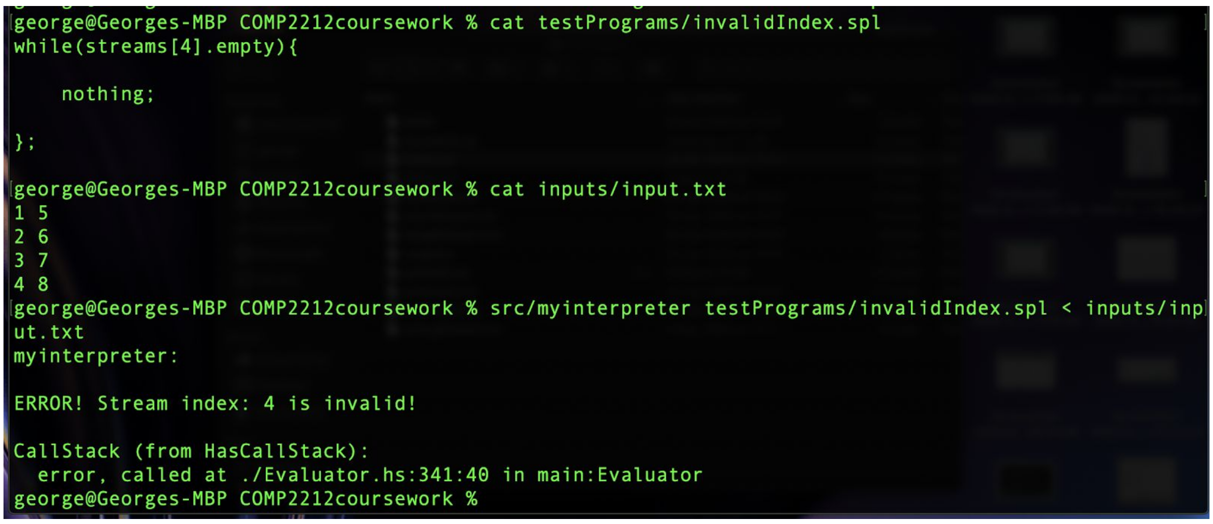
Only allowing integer values in the stream, not allowing any kind of symbols:


```
george@Georges-MBP COMP2212coursework % cat testInputs/symbols.txt
( *
) %
[ $
] £
george@Georges-MBP COMP2212coursework % src/myinterpreter programs/pr1.spl < testInputs/symbols.txt
myinterpreter: Invalid values detected in stream input, only Ints are permitted!
CallStack (from HasCallStack):
  error, called at ./IOReader.hs:22:65 in main:IOReader
```

Checking the correct number of streams has been provided, e.g. here Program 1 references 2 streams:

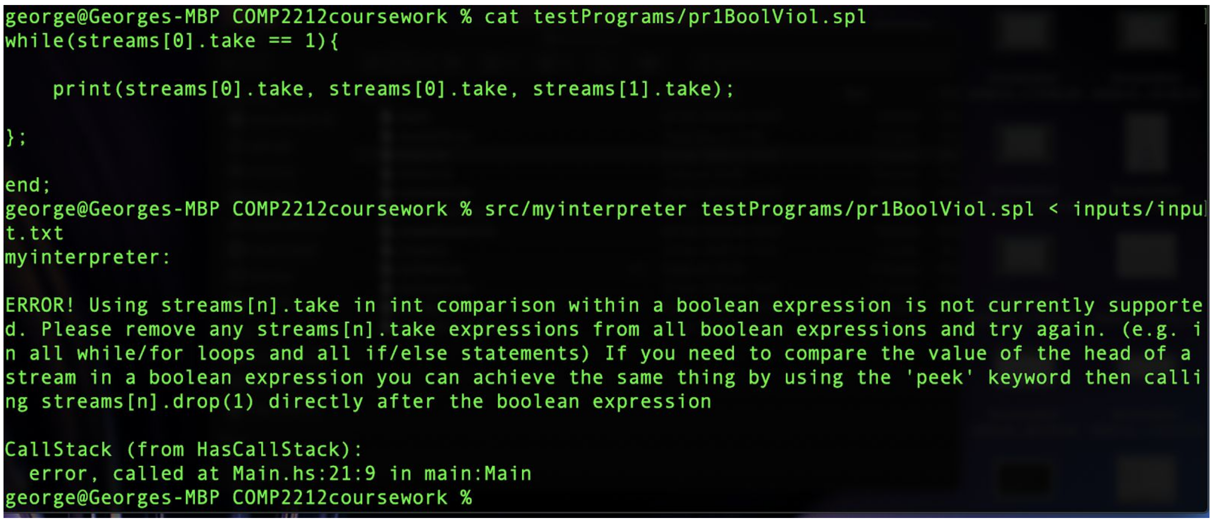

```
george@Georges-MBP COMP2212coursework % cat testInputs/singleStream1.txt
1
2
3
4
george@Georges-MBP COMP2212coursework % src/myinterpreter programs/pr1.spl < testInputs/singleStream
1.txt
myinterpreter:

ERROR! Incorrect number of streams provided! This program was expecting: 2 streams, but received: 1.
 Please make sure you have provided the correct number of streams in your stream data file and try a
gain

CallStack (from HasCallStack):
  error, called at Main.hs:38:13 in main:Main
```

Not allowing referencing of streams that don't exist:



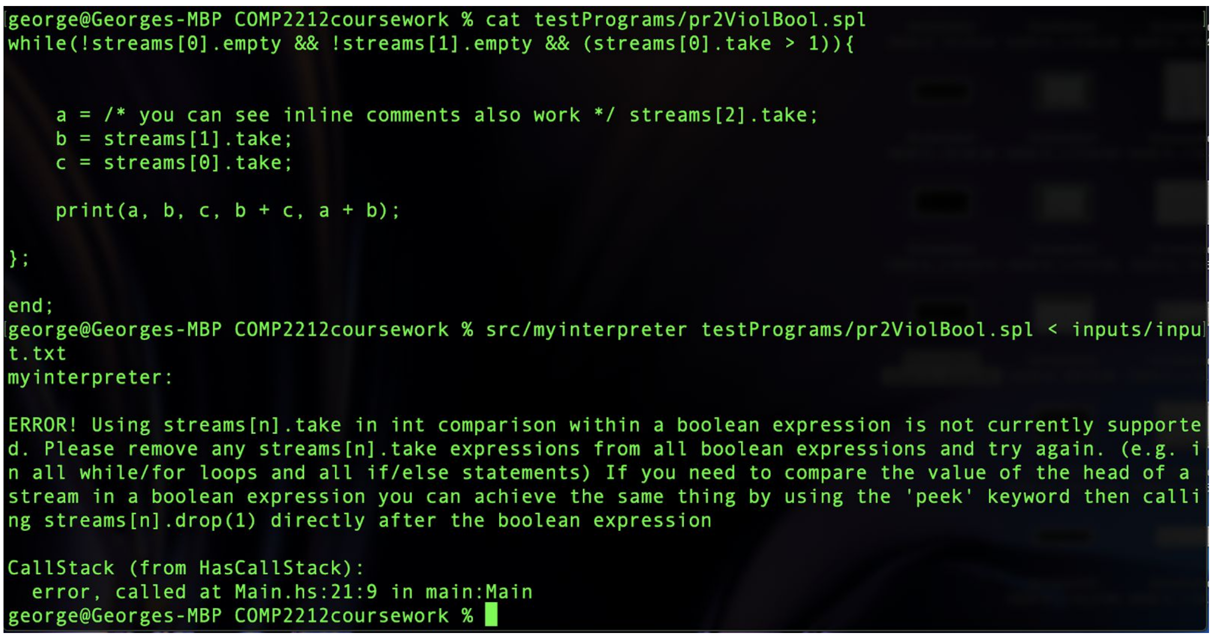
```
george@Georges-MBP COMP2212coursework % cat testPrograms/invalidIndex.spl
while(streams[4].empty){

    nothing;

};

george@Georges-MBP COMP2212coursework % cat inputs/input.txt
1 5
2 6
3 7
4 8
george@Georges-MBP COMP2212coursework % src/myinterpreter testPrograms/invalidIndex.spl < inputs/inp
ut.txt
myinterpreter:

ERROR! Stream index: 4 is invalid!

CallStack (from HasCallStack):
  error, called at ./Evaluator.hs:341:40 in main:Evaluator
george@Georges-MBP COMP2212coursework %
```

Not allowing the use of "take" in int value comparisons within a boolean expression:



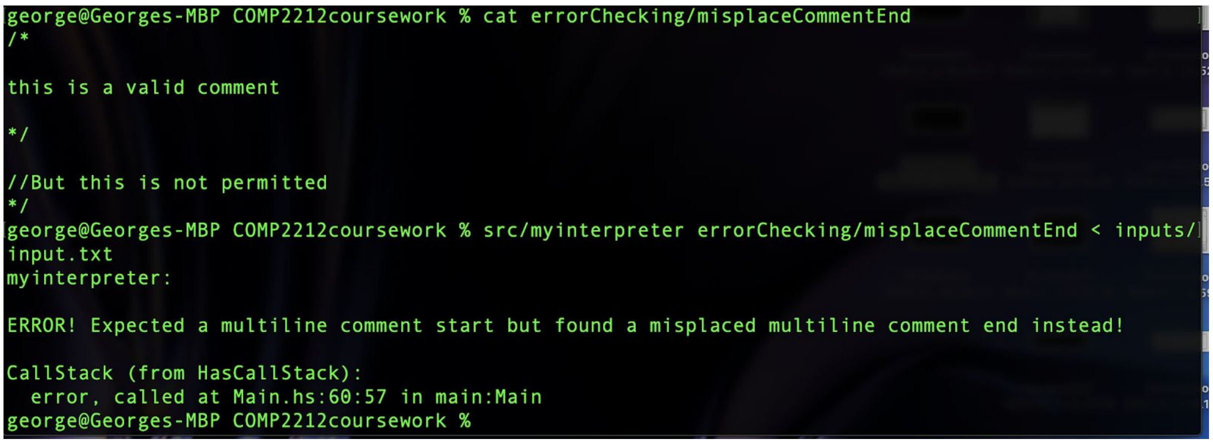
```
george@Georges-MBP COMP2212coursework % cat testPrograms/pr1BoolViol.spl
while(streams[0].take == 1){

    print(streams[0].take, streams[0].take, streams[1].take);

};

end;
george@Georges-MBP COMP2212coursework % src/myinterpreter testPrograms/pr1BoolViol.spl < inputs/inpu
t.txt
myinterpreter:

ERROR! Using streams[n].take in int comparison within a boolean expression is not currently supporte
d. Please remove any streams[n].take expressions from all boolean expressions and try again. (e.g. i
n all while/for loops and all if/else statements) If you need to compare the value of the head of a
stream in a boolean expression you can achieve the same thing by using the 'peek' keyword then calli
ng streams[n].drop(1) directly after the boolean expression

CallStack (from HasCallStack):
  error, called at Main.hs:21:9 in main:Main
george@Georges-MBP COMP2212coursework %
```

Not allowing the use of "take" in int value comparisons within a boolean expression:



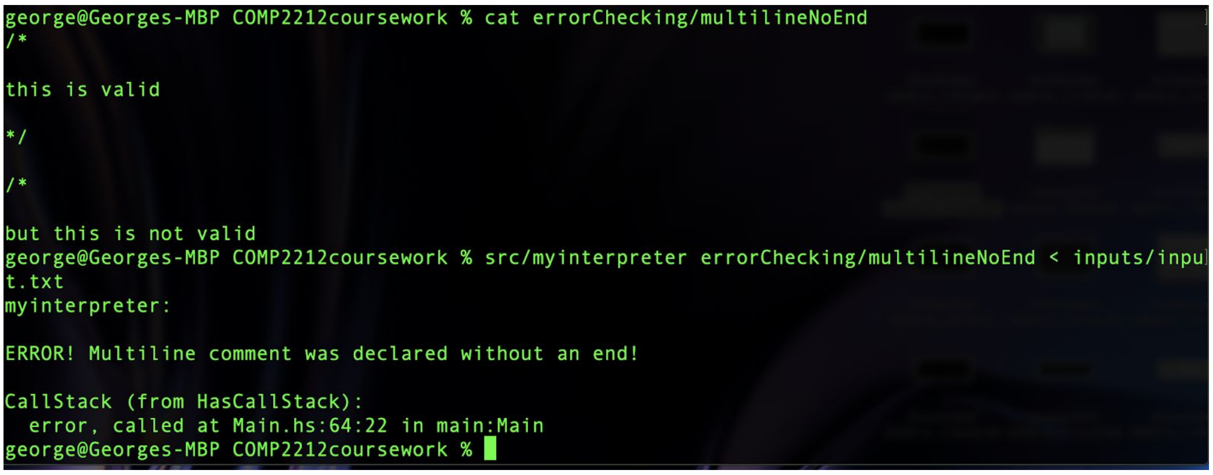
```
george@Georges-MBP COMP2212coursework % cat testPrograms/pr2ViolBool.spl
while(!streams[0].empty && !streams[1].empty && (streams[0].take > 1)){


    a = /* you can see inline comments also work */ streams[2].take;
    b = streams[1].take;
    c = streams[0].take;

    print(a, b, c, b + c, a + b);

};

end;
george@Georges-MBP COMP2212coursework % src/myinterpreter testPrograms/pr2ViolBool.spl < inputs/inpu
t.txt
myinterpreter:

ERROR! Using streams[n].take in int comparison within a boolean expression is not currently supporte
d. Please remove any streams[n].take expressions from all boolean expressions and try again. (e.g. i
n all while/for loops and all if/else statements) If you need to compare the value of the head of a
stream in a boolean expression you can achieve the same thing by using the 'peek' keyword then calli
ng streams[n].drop(1) directly after the boolean expression

CallStack (from HasCallStack):
  error, called at Main.hs:21:9 in main:Main
george@Georges-MBP COMP2212coursework %
```

Misplaced multiline/inline comment end:



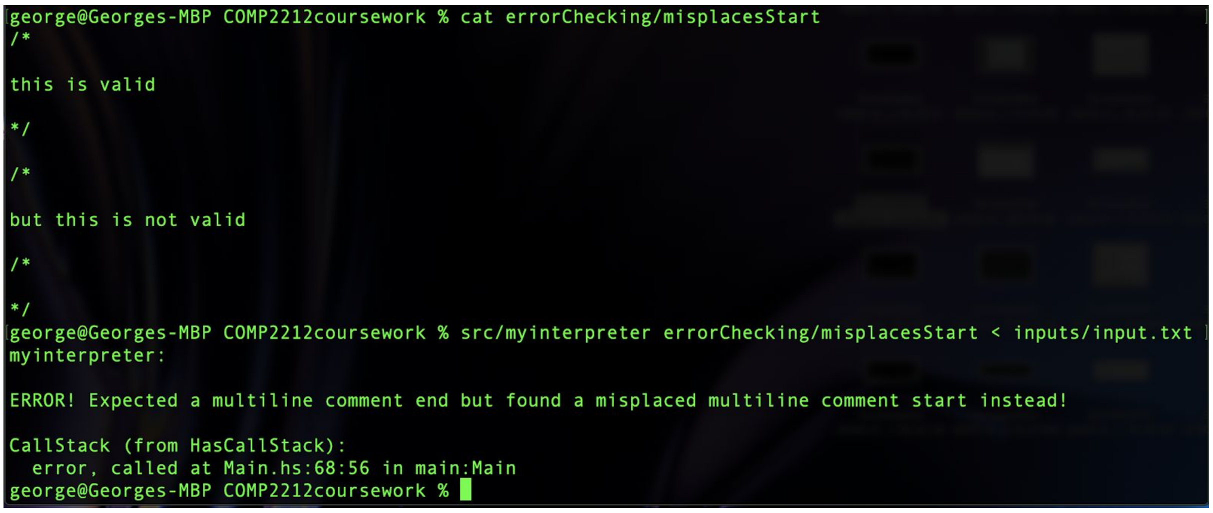
```
george@Georges-MBP COMP2212coursework % cat errorChecking/misplaceCommentEnd
/*

this is a valid comment

*/

//But this is not permitted
*/
george@Georges-MBP COMP2212coursework % src/myinterpreter errorChecking/misplaceCommentEnd < inputs/
input.txt
myinterpreter:

ERROR! Expected a multiline comment start but found a misplaced multiline comment end instead!

CallStack (from HasCallStack):
  error, called at Main.hs:60:57 in main:Main
george@Georges-MBP COMP2212coursework %
```

Not allowing multiline/inline comments to be declared without an end:



```
george@Georges-MBP COMP2212coursework % cat errorChecking/multilineNoEnd
/*

this is valid

*/

/*

but this is not valid
george@Georges-MBP COMP2212coursework % src/myinterpreter errorChecking/multilineNoEnd < inputs/inpu
t.txt
myinterpreter:

ERROR! Multiline comment was declared without an end!

CallStack (from HasCallStack):
  error, called at Main.hs:64:22 in main:Main
george@Georges-MBP COMP2212coursework %
```

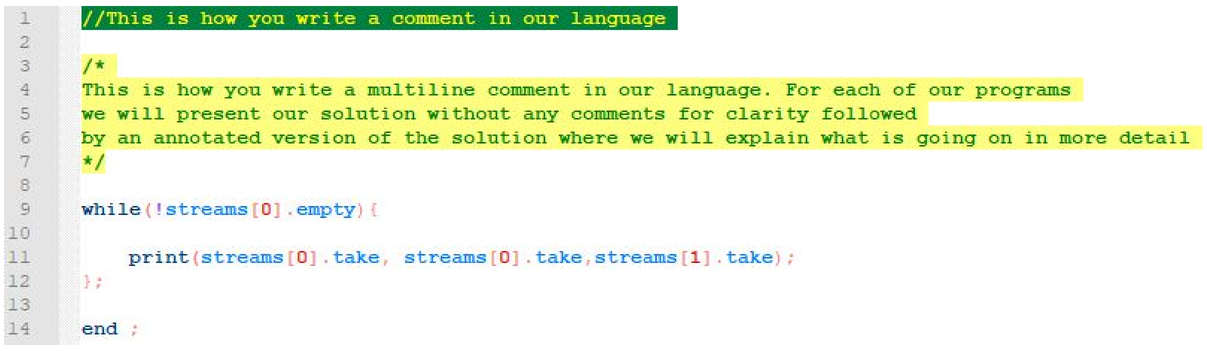
Misplaced multiline/inline comment start:



```
george@Georges-MBP COMP2212coursework % cat errorChecking/misplacesStart
/*

this is valid

*/

/*

but this is not valid

/*

*/
george@Georges-MBP COMP2212coursework % src/myinterpreter errorChecking/misplacesStart < inputs/input.txt
myinterpreter:

ERROR! Expected a multiline comment end but found a misplaced multiline comment start instead!

CallStack (from HasCallStack):
  error, called at Main.hs:68:56 in main:Main
george@Georges-MBP COMP2212coursework %
```

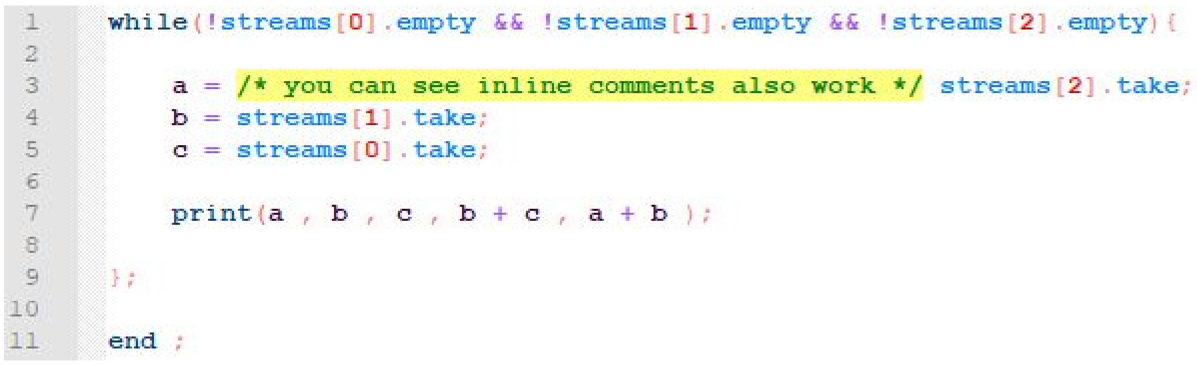
Syntax highlighting

Problem 1:



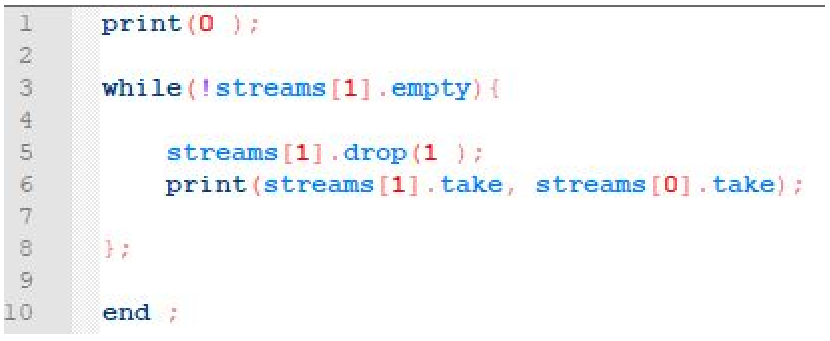
```
//This is how you write a comment in our language

/*
This is how you write a multiline comment in our language. For each of our programs
we will present our solution without any comments for clarity followed
by an annotated version of the solution where we will explain what is going on in more detail
*/

while(!streams[0].empty){

    print(streams[0].take, streams[0].take,streams[1].take);
};

end ;
```

Problem 2:



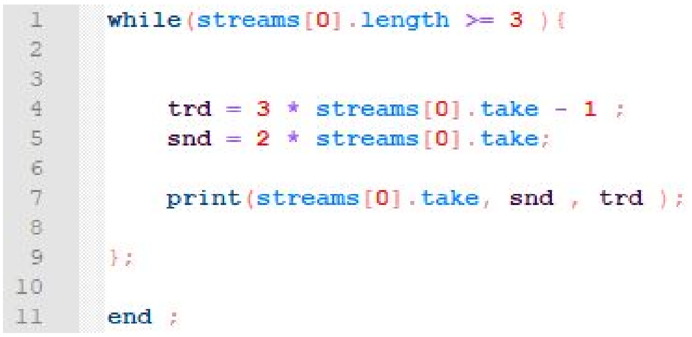
```
while(!streams[0].empty && !streams[1].empty && !streams[2].empty){

    a = /* you can see inline comments also work */ streams[2].take;
    b = streams[1].take;
    c = streams[0].take;

    print(a , b , c , b + c , a + b );

};

end ;
```

Problem 3:



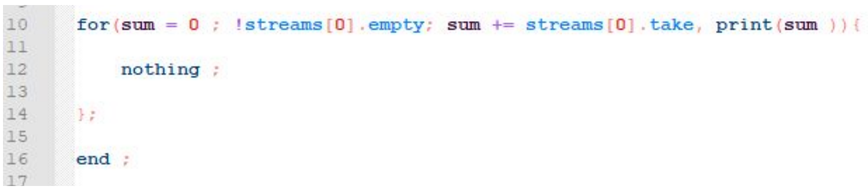
```
print(0 );

while(!streams[1].empty){

    streams[1].drop(1 );
    print(streams[1].take, streams[0].take);

};

end ;
```

Problem 4:



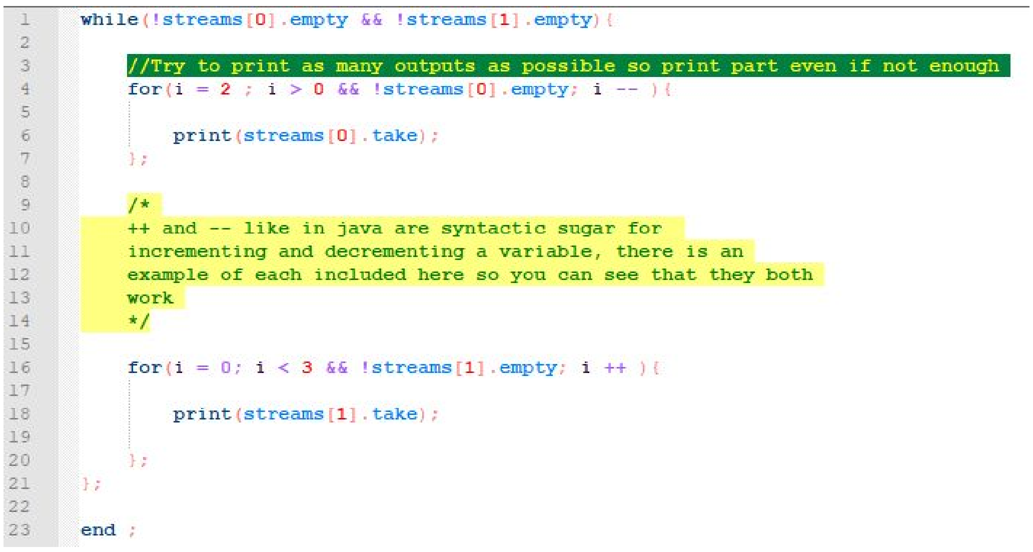
```
while(streams[0].length >= 3 ){


    trd = 3 * streams[0].take - 1 ;
    snd = 2 * streams[0].take;

    print(streams[0].take, snd , trd );

};

end ;
```

Problem 5:



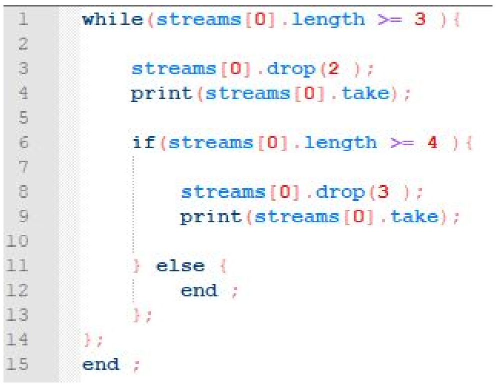
```
for(sum = 0 ; !streams[0].empty; sum += streams[0].take, print(sum )){

    nothing ;

};

end ;
```

Problem 6:



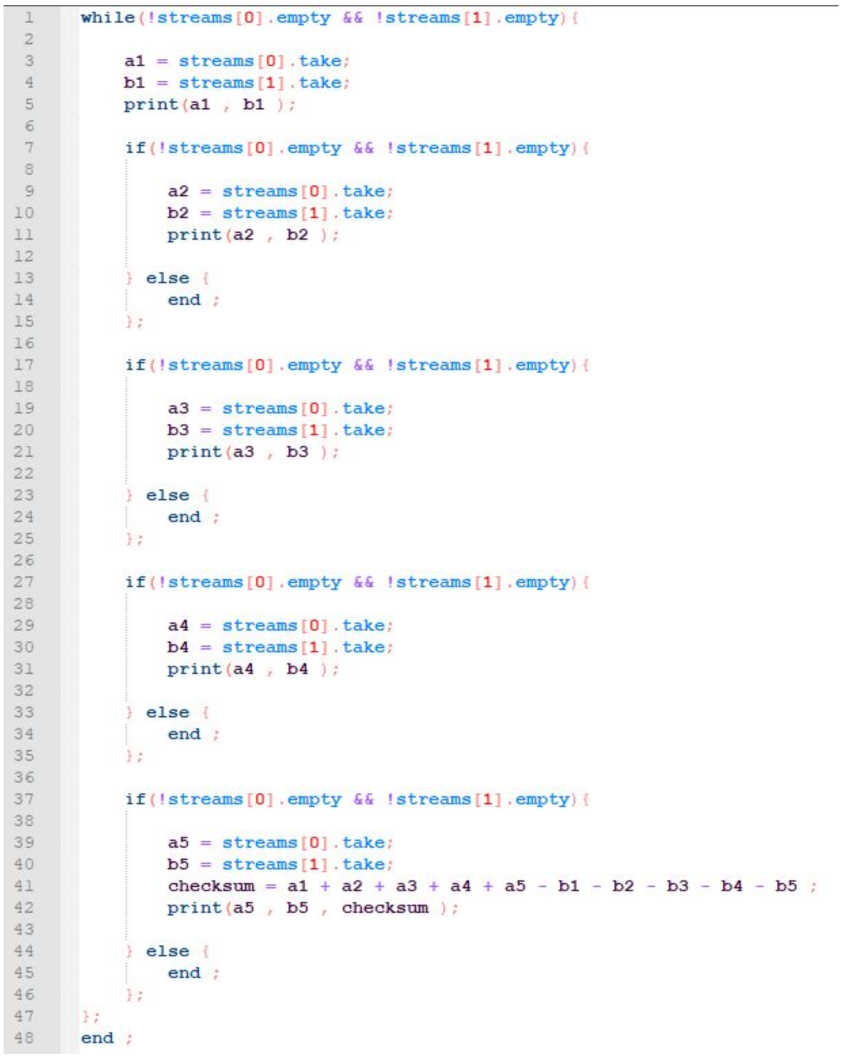
```
while(!streams[0].empty && !streams[1].empty){

    //Try to print as many outputs as possible so print part even if not enough
    for(i = 2 ; i > 0 && !streams[0].empty; i -- ){

        print(streams[0].take);
    };

    /*
    ++ and -- like in java are syntactic sugar for
    incrementing and decrementing a variable, there is an
    example of each included here so you can see that they both
    work
    */

    for(i = 0; i < 3 && !streams[1].empty; i ++ ){

        print(streams[1].take);

    };
};

end ;
```

Problem 7:



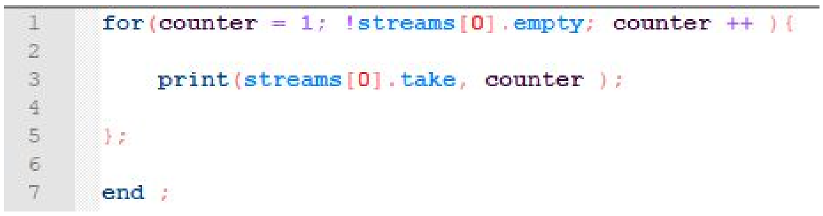
```
while(streams[0].length >= 3 ){

    streams[0].drop(2 );
    print(streams[0].take);

    if(streams[0].length >= 4 ){

        streams[0].drop(3 );
        print(streams[0].take);

    } else {
        end ;
    };
};
end ;
```

Problem 8:



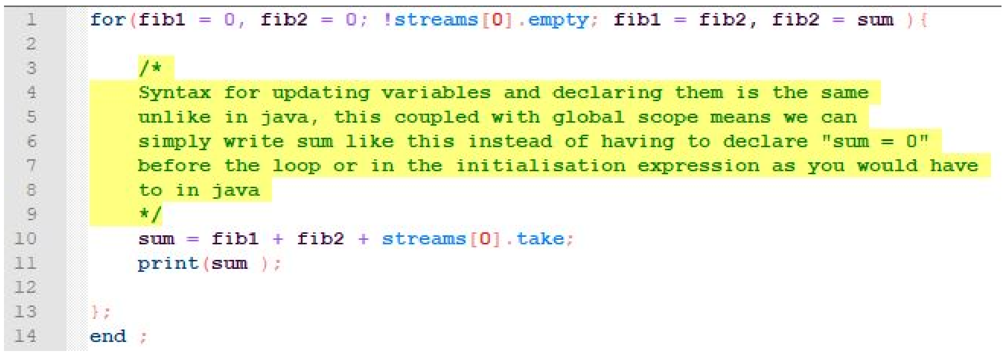
```
while(!streams[0].empty && !streams[1].empty){

    a1 = streams[0].take;
    b1 = streams[1].take;
    print(a1 , b1 );

    if(!streams[0].empty && !streams[1].empty){

        a2 = streams[0].take;
        b2 = streams[1].take;
        print(a2 , b2 );

    } else {
        end ;
    };

    if(!streams[0].empty && !streams[1].empty){

        a3 = streams[0].take;
        b3 = streams[1].take;
        print(a3 , b3 );

    } else {
        end ;
    };

    if(!streams[0].empty && !streams[1].empty){

        a4 = streams[0].take;
        b4 = streams[1].take;
        print(a4 , b4 );

    } else {
        end ;
    };

    if(!streams[0].empty && !streams[1].empty){

        a5 = streams[0].take;
        b5 = streams[1].take;
        checksum = a1 + a2 + a3 + a4 + a5 - b1 - b2 - b3 - b4 - b5 ;
        print(a5 , b5 , checksum );

    } else {
        end ;
    };
};
end ;
```

Problem 9:



```
for(counter = 1; !streams[0].empty; counter ++ ){

    print(streams[0].take, counter );

};

end ;
```

Problem 10:



```
for(fib1 = 0, fib2 = 0; !streams[0].empty; fib1 = fib2, fib2 = sum ){

    /*
    Syntax for updating variables and declaring them is the same
    unlike in java, this coupled with global scope means we can
    simply write sum like this instead of having to declare "sum = 0"
    before the loop or in the initialisation expression as you would have
    to in java
    */
    sum = fib1 + fib2 + streams[0].take;
    print(sum );

};
end ;
```