Jh11g20 PLC Language Report

Language name: SimpleStream

This is a programming language intended specifically for the processing and output of 1 or more data streams of integers.
The main priority of its design is simplicity, and closeness to written english.
It is an imperative language with strong, static typing and explicit variable declaration. It supports lazy evaluation in places, and uses a mix of lazy and eager evaluation.

Syntax

A program must begin with a ‘{‘ character and it must end with a ‘}’ character.

The syntax is very simple. Functions, types, and values are generally written as words in all capital letters. To stop confusion between function names and variables, variables are required to begin with a lowercase letter (this incentivises using lowerCamelCase).

There are 4 types: BOOL, INT, STRING, ARRAY

BOOL can consist of either TRUE or FALSE

INT is simply an integer, either positive or negative

STRING is written as “characters to write” like most languages. Note that the allowed characters for a string consist of all characters except a newline.

ARRAY is an array/list of INT. Essentially a data stream. They are stored as a haskell list and so can change size and their size need not be declared.

The actions of the language

The language consists of a series of actions, each one ending with a ‘;’ character much like java.
The language ignores whitespace and newlines so theoretically a whole program could fit onto one line, this is definitely not recommended for readability’s sake.
The language also does not care about indentation though the user can very much choose to use indentation to improve the readability of the code, it is not required, allowing them more freedom.

The actions of the language include:
Conditionals:

IF (Expression (must eval to boolean))
Action;
Action;
Action;
ENDIF;

The condition for the if statement (and indeed all other conditional actions in the language) must be contained within brackets, and the ‘;’ only goes on the end of ENDIF statement, not at the end of the condition. The if statement is evaluated as one whole action.

IF (Expression (must eval to boolean))
Action;
Action;
ELSE
Action
ENDIF;

Is for an If/else condition. Once again, the ‘;’ only goes at the ENDIF marker.

Loops
WHILE (Expression (must eval to boolean))
Action;
Action;
ENDLOOP;

Syntax is essentially identical to the if statements.

FOR (Expression (must eval to int))
Action;
Action;
ENDLOOP;

For loops specifically do the specified actions a set number of times specified by the value within the brackets.

Note that there is no scoping in this language, all variables used should be declared before entering a conditional block or loop, ideally in the beginning of the program.

Variables
The language requires type declaration of new variables, you also cannot re-declare an existing variable with a new type.
Variable names must begin with a lowercase letter and can follow with any number of lower or uppercase letters, numbers, “_” or “ ‘ “

Declaring a new variable is as follows:

TYPE:varName =* value;
For example:
INT:variable_1 =* 5+6;

Note that the assignment operator is '=*' this is because the equivalence operator is simply = this is to make the operators a bit closer to familiar english/maths conventions and to show that assignment is an act of actively making a variable equivalent to something.

To update a variable you simply write:

varName =* value;

There is some syntactic sugar too, writing 'varName++;' is equivalent to writing 'varName =* varName + 1;'

ADD
The last action is ADD which is the action to add an int to the end of an ARRAY
The syntax is: ADD arrayName value;

Expressions

Values
Expressions consist of all the things that you can write within an action. They include:
Base values, such as: 12, "hello", FALSE, [2,3,4], or just the name of a variable.

Mathematics
The language has simple mathematical operations of +, -, *, / as well as <, >, <=, >=, != (is not equal to), =
It essentially follows the same standard as most languages, except '=' means equivalence.
Addition and subtraction are left associative, multiplication and division are right associative. All operations follow BIDMAS.
Note that only Integers are allowed, so division will only produce the number of times the number fits within the other number, it will not include the remainder.

Boolean logic
There is also Boolean logic, such as: NOT, AND, OR, XOR written as you naturally would:
NOT naturally binds tighter than the others.
XOR binds slightly tighter than AND and OR and is non-associative
AND and OR are left associative

Examples: x AND NOT y a OR b NOT c XOR d

Array value-at-index retrieval

Then there is array retrieval, which is written as ‘arrayName AT value’ to retrieve the number at an index of the array. As arrays in this language only contain int values, this will always evaluate to an int, except in one case; IN.

IN, the input value
IN is the languages value for the input streams, it is a 2d array of arrays of ints representing each column of the input. For example ‘IN AT 0’ would evaluate to an array of the 1st column of the input streams.

Length of arrays
The last one is LENGTH which evaluates to the length of a given array. An example: LENGTH varName.

Lazy length comparison allows infinite stream processing
One note about LENGTH, the language deals with potentially unbounded data streams, and as such evaluates lazily, if you wanted to check if there is values left in the stream for however many values ahead you need you might write ‘LENGTH stream > index1 + 5’ for example. When writing a comparison expression using length such as the example, the language will not actually evaluate the full length of the array, it will only confirm if the array is longer than the specified value, which, paired with the lazy storage and reading of inputs by the interpreter, allows for computation and output whilst input is still being read, as the interpreter will only read the input stream when it needs values from it for the computation.

Brackets
As you might expect, the language allows for brackets to be put around expressions or actions to group them as you naturally might want to.

Comments
You can make a comment by preceding a line with ‘--’ the same as you do in haskell.

Associativity order (from highest precedence to lowest precedence, nonassociative unless specified)
‘{‘ ‘}’ : ’(‘ ‘)’ : int string TRUE FALSE : variables array IN : LENGTH : ‘-’ (on its own to signify a negative value, left assoc) : ‘*’ ‘/’ (right assoc) : ‘+’ ‘-’ (left assoc) : ‘<’ ‘>’ ‘<=’ ‘>=’ ‘!=’ ‘=’ : NOT (right assoc) : XOR : AND OR : AT (left assoc) : ‘=*’ ADD RETURN : IF WHILE FOR ELSE ENDIF ENDLOOP : BOOL INT STRING ARRAY : ‘;’

Execution

Lexing and parsing

The interpreter will lex the program file then parse it, at either of these stages if an error is found, there will be an error message informing you of the line and column number where the

lexical or parse error occurred.

```
C:\Users\John\Documents\Southampton Computer science\Year 2\Programming concepts\Submission Files>Spl.exe pr4.spl < ExampleInput4.txt
Parse error at line:column 8:12
CallStack (from HasCallStack):
  error, called at .\ProjectGrammar.hs:1418:21 in main:ProjectGrammar
```

Type checking

It will then perform a type check. Essentially, it evaluates the program, but using the types instead of the actual values, this produces a list of types to correspond with the list of actions, the list is not output. But if an error occurs it will be output to stderr.
If there is a type error, it will be notified and outputted with the particular part of the parse tree that caused the issue.

```
Type error PWhile (PAdd (PLength (PVar "var1")) (PVar "index1")) [PAddToIntArray (PVar "out") (PAdd (PGetIntValueAt (PVa
r "var1") (PVar "index1")) (PGetIntValueAt (PVar "out") (PSub (PVar "index1") (PInt 1)))),PReturn (PGetIntValueAt (PVar
"out") (PVar "index1")),PUpdateVar (PVar "index1") (PAdd (PVar "index1") (PInt 1))]
CallStack (from HasCallStack):
  error, called at .\ProjectTypes.hs:93:18 in main:ProjectTypes
```

Such as this error which essentially tells you there is a type error in the condition of the while loop (you can see that it evaluates to a number and not a boolean). There is extra information because it will display the entirety of the while's parse tree, but you need only look at the first bit because if it were an error within the actions of the while statement, those actions would be outputted as error, not the whole while statement.

Evaluation

After that it will evaluate the program, again, if any errors occur, they will be output to the stderr. Most evaluation errors will essentially be standard haskell errors such as an index being out of bounds for an array or an infinite loop. For example:

```
Spl.exe: ProjectEval.hs:(147,1)-(187,80): Non-exhaustive patterns in function value
```

Evaluation occurs as a series of steps evaluating each action. The parse output is a list of trees consisting of each action. The evaluator uses states that are of type [(String,Expr)] which is essentially a list of all the variables that have been declared and their values. The main interpreter function recursively moves through the list of actions evaluating each action with an action evaluator function, using the state output of one action as the state input for the next one. Because the program must output during runtime, this is an IO function, it uses a stateT type wrapper to be able to thread the environment as a state through the functions which allows pure functions to be called with the state output of the impure evaluation function as an argument.

The action evaluator will process the action and for expression evaluation will call a pure function to recursively evaluate the expression using big step semantics. This is because the only output that needs to be done during runtime is at the end of actions.

The eventual end state is ultimately ignored as the only output needed will have already been outputted by the end of runtime.