COMP2212 Programming Language Concepts
aeg3g18 Adam Gafar
mas1g18 Morgan Shipsides

Interpreter/Language Report

Language overview

Programs in our language are split into two parts: a preamble in which variables can be defined and values may be output to `stdout`; and a main loop in which all commands are permitted, and is executed repeatedly on buffered blocks of the input stream until the program exhausts the input data. Other than handling buffered values and outputting, commands are simple and limited to simple arithmetic operations and variable assignment.

The interpreter that we have constructed produces basic error messages and warnings that may assist the programmer in debugging attempts, and executes programs using slightly different logic and state models for each of the two program stages.

Grammar

The grammar of our language is expressed in the following *Backus-Naur Form* (BNF) notation:

```
<Program> ::= ‘End’ | <Command> <Program>

<Command> ::= ‘Out’ <Exp> | ‘SetVar’ '~'<int> <Exp> | ‘Loop’ | ‘Buffer’
<Exp> <Exp>

<Exp> ::= 'L'<int> | ‘Release’ <Exp> <Exp> | ‘Add’ <Exp> <Exp> | ‘Sub’
<Exp> <Exp> | ‘Mul’ <Exp> <Exp> | ‘Div’ <Exp> <Exp> | '(' <Exp> ')' |
‘Var’ <int>

<int> ::= ‘-’ <number> | <number>
<number> ::= <number><num> | <num>
<num> ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

Syntax

The syntax mostly follows the above grammar, with a few additional restrictions to ensure the separation of the preamble and main execution loop. In the preamble, the programmer is limited to only the `SetVar` and `Out` commands so none of the stream can be consumed before the loop begins; the main loop permits all of the following commands, however:

`()`			*Used to group for ordering*
`Add`	[value 1]	[value 2]	*Add value 1 to value 2*
`Sub`	[value 1]	[value 2]	*Subtract value 1 from value 2*
`Mul`	[value 1]	[value 2]	*Multiply value 1 by value 2*
`Div`	[value 1]	[value 2]	*Divide value 1 by value 2*
`Out`	[value]		*Output value to* `stdout`
`Buffer`	#numberOfVals	@streamNum	*Buffer x values from stream y*
`SetVar`	~varID	[value]	*Set variable x to value y*

`Release`	#valIndex	@bufferNum	*Returns value at index x from buffer y*
`Var`	varID		*Returns the value of variable x*

`Loop`	Signifies the end of preamble and beginning of the looped program. Every program's code requires a single instance of this command; in the case of no preamble this is the first statement.
`End`	Signifies the end of the program; the programmer is not permitted to use this command as it is added internally (within the interpreter) to terminate recursion of the `<Program>` grammar.
`>>`	Precedes comments, anything subsequent on the line is ignored by the interpreter.

Language features

To reiterate, programs written in our language are comprised of two stages, namely the *preamble* and *execution loop*. The preamble is intended to be used to initialise variables and output values that are to be appended to the front of the stream (i.e. for results to the problems that did not require reading any data). The main loop permits all commands to the programmer and is executed repeatedly until the interpreter reaches its exit condition.

At the end of the loop the buffer is reset, clearing it of any values that were unused in the current iteration. This is the only time values are removed from the buffer in the execution of the program; the function of `Release` is modelled to return the values that were buffered without changing program state and does not release them from memory.

Execution stops when a command that cannot be performed is reached, with the exception of `Buffer` when insufficient data has been loaded to buffer from. In said case, the program simply loads as many values as it can up to the requested quantity and finishes if the end of the input stream is found.

Scoping

The language has a single scope; all variables are "global" from creation. This extends across the preamble and execution loop, allowing for variables to be both initialised and used in the preamble to form the initial values of the output stream, and then changed within the execution loop to achieve the correct outputs for the problem set with minimal inconvenience to the programmer.

Convenience

For visual clarity, the programmer can prefix an `int` to indicate its meaning under certain scenarios (the interpreter will not discriminate if they choose not to). Both arguments of the `Buffer` and `Release` commands can be appropriately prefixed to this end (e.g. `Buffer #2 @1` highlighting that it buffers two values from the first input stream). `~` can be used to prefix variables (e.g. `SetVar ~1 L1`) in the same way, with `L` indicating an integer literal for any `<Exp>` as shown.

Type checking and error messages

Our language only supports the `int` data type, and we therefore have not implemented any type checking. However, we have implemented syntax checking that will give warnings if supplied code has multiple `Loop` commands, or if a variable is used before it has been set. These warnings are printed to **`stderr`** (execution is still attempted, though it is unlikely to succeed).

Errors found during lexing or parsing result in simple error messages being displayed, with the same occurring with any errors encountered during execution (along with the program exiting).

Influences

To some extent, our language could be viewed as having been inspired by *Pascal*. There is a major difference between the two languages though, in that in our language commands can be executed in the preamble whereas in Pascal only variable type initialisations are permitted.

Program state

The following table lists the five entities comprising the state of a program in this language:

Name	Type	Description
`loads`	List of `int` lists	Values read in from the text input stream.
`buff`	List of `int` lists	Values currently loaded from **`loads`** into the buffer.
`vars`	List of `int` pairs	Variable names (integer IDs) paired with their values.
`loop`	`<Program>`	All commands in the main execution loop of the program.
`instr`	`<Program>`	Current instructions/commands yet to be executed in this loop iteration.

The program state can be transformed in the following ways during the main execution loop:

- `SetVar` can assign values to new or existing variables to add to **`vars`**.
- `Buffer` removes values from **`loads`** and copies them to the requested **`buff`** list, and can cause values to be loaded from the input data stream into **`loads`** first if it doesn't contain the requested quantity of values to buffer.
- Upon command execution, the current instruction in **`instr`** is advanced – if the end of the loop is reached then it is reset to the start of **`loop`** and **`buff`** is cleared.

The preamble follows a different design as it does not require the entire state – only `Out` and `SetVar` are available for use; the input stream cannot be accessed and thus neither **`loads`** nor **`buff`** can change. Since none of the commands from the main loop are executed in the preamble, **`loop`** and **`instr`** are also unaffected. Variables can be freely assigned however, and as such it is permitted to change **`vars`** until exhausting all statements, after which it passes its final state to form the initial state of the main loop.