Assignment 1

Deadline:	Hand in by 5pm on Friday 1st April 2022
Evaluation:	10 marks – which is 10% of your final grade
Late Submission:	1 mark off per day late
Work:	This assignment is to be done **individually** – your submission may be checked for plagiarism against other assignments and against Internet repositories. If you adapt material from the internet you must acknowledge your source.
Purpose:	To reinforce ideas covered in the lectures for understanding and using imperative programming languages and language design concepts.

Problem to solve:

Write a parser/interpreter for a simple, imperative string-processing language.

Requirements:

The interpreter must read input from the standard input (`stdin`) and write output to the standard output (`stdout`) and should read statements from the user one at a time and perform the associated action immediately. If invalid input is provided, the interpreter should attempt to recover and try to provide as useful an error message as possible.

The grammar for the strings processing language is given below:

```
program     := { statement }
statement   := 'append' identifier expression ';'
             | 'list' ';'
             | 'exit' ';'
             | 'print' expression ';'
             | 'printlength' expression ';'
             | 'printwords' expression ';'
             | 'printwordcount' expression ';'
             | 'set' identifier expression ';'
             | 'reverse' identifier ';'
expression  :=  value { '+' value }
value       := identifier | constant | literal
constant    := 'SPACE' | 'TAB' | 'NEWLINE'
literal     := '"' { letter | digit | punctuation } '"'
identifier  := letter { letter | digit }
letter      := 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z'
digit       := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
punctuation := '.' | ',' | ':' | ';' | '?' | '!' | '"' | ...
```

The intended behavior of each instruction is given in the following table:

Command	Parameters	Behaviour
`append`	**`identifier expression`**	Evaluate the expression and append it to the contents of the variable.
`list`		List all variables and their contents.
`exit`		Exit the interpreter
`print`	**`expression`**	Evaluate and print the expression
`printlength`	**`expression`**	Evaluate the expression and print its length
`printwords`	**`expression`**	Evaluate the expression and print the individual words
`printwordcount`	**`expression`**	Evaluate the expression and print the number of words
`set`	**`identifier expression`**	Set (create if necessary) the contents of the variable to the expression
`reverse`	**`identifier`**	Reverse the order of the words in the contents of the variable.

Sample input might be:

```
set one "The cat";
set two "sat on the mat";
set sentence one + SPACE + two;
append sentence " by itself.";
print sentence;
printwordcount sentence;
printwords sentence;
printlength sentence;
list;
reverse one;
print one;
exit;
```

From which output **like** the following could be expected:

```
---------------------------------------
159.341 2022 Semester 1, Assignment 1
Submitted by Rick Deckard, 20191187
---------------------------------------
The cat sat on the mat by itself.
Wordcount is: 8
Words are:
The
cat
sat
on
the
mat
by
itself.
Length is: 33
Identifier list (3):
one: "The cat"
two: "sat on the mat"
sentence: "The cat sat on the mat by itself."
cat The
```

Notes:

Your program can just read from `stdin` and write to `stdout`, it doesn't need to open and close any files. You might want to use `stderr` for error messages if it finds input situations it cannot handle or runs out of resources.

One of the first things you should do in your program design is decide what **data structures** you will use and how to organise them. You will need some way of storing a **symbol table** which will store all the **variables** in your program (and provide a way to access their contents). This language has only **one implicit type** (a string), so your symbol table does not need to store any information about the type of the variable.

You should also think about what **functions/procedures** you will need. You may want to consider a **recursive descent parser** as a base for your interpreter. Since each statement starts with a command, you could start by writing a function to work out which command it is. Based on the command, your parser will know what terms are expected to follow.

For example - if you parse the command '`list`' then you expect it to be followed by a '`;`'. If it is not followed by a semicolon then an error has occurred. Likewise, if you parse the command '`set`' you will then need to parse an *identifier name* and then an *expression* followed by a semicolon.

The language has only specified a single operator, the binary + operator for concatenation. Think of + as a function that takes two strings and returns a third which is the concatenation of the two arguments. This operator is associative (it does not matter whether you resolve the concatenations left-to-right or right-to-left. Expressions may have any number of terms connected with + operators.

The language specifies three constants:

```
SPACE  as  " "
TAB as "\t"
NEWLINE  as "\n"
```

These could be parsed separately by your interpreter or you could treat them as identifiers and store them as variables in your symbol table (you would need to have some way of preventing the user from changing their values).

Decisions:
There are some features that have been left (intentionally) unspecified.

- How are you going to deal with special characters inside a string literal? In particular, how are you going to deal with double-quotes inside a string?
- How is your interpreter going to respond when the user enters invalid input? Your program should try to provide some type of error handling and should not just crash (a common approach is just to read and ignore all input until it reaches the next ';').
- The definition of a words has also been left slightly vague – should a word be separated by a space or does any non-letter character count to separate words?

These decisions are left up to you to make but you should document them within your assignment comments.

Some sample input (along with *possible* output) will be provided on the stream site that you can use to test your interpreter.

You **must** follow the next two specifications in **each and every assignment for this course**
1. Place the following comments at the top of your program code and **provide the appropriate information**:

```
/* Family Name, Given Name, Student ID, Assignment number, 159.341 */
/* explain what the program is doing . . . */
```

2. Ensure that your program prints this information to the console. You might use code like:

```
"--------------------------------------"
"  159.341 Assignment 1 Semester 1 2022  "
"  Submitted by: Rick Deckard, 20191187  "
"--------------------------------------"
```

Hand-in: Submit **your program and documentation (a zip file is acceptable)** electronically through the form on the stream site.
Marks will be allocated for: correctness, fitness of purpose, sensible use of data structures and algorithms, utility, style, use of sensible **comments and program documentation**, and general elegance. Good comments will help me to award you marks even if your code is not quite perfect.

If you have any questions about this assignment, please ask the lecturer.