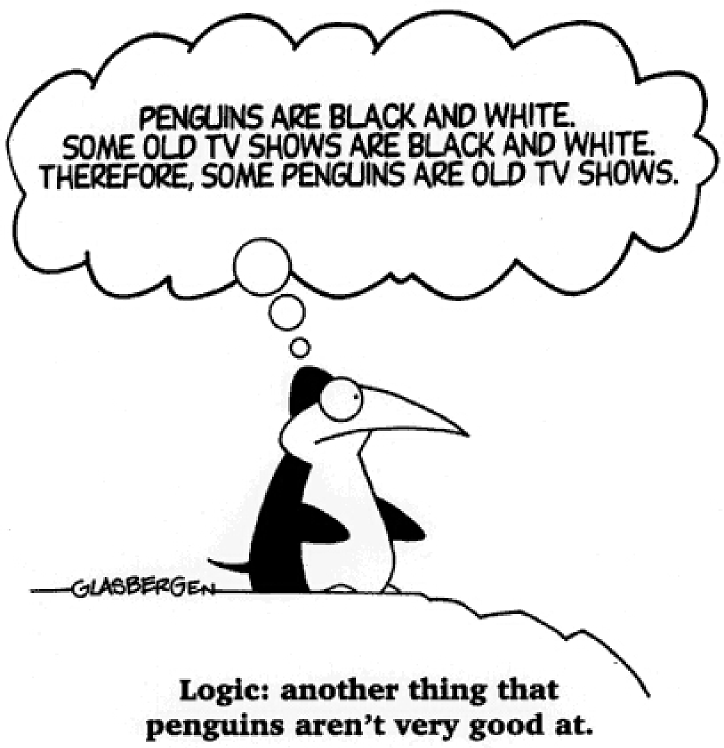
ENGINEERING TRIPOS PART IIA

DPO PROJECT GF2

SOFTWARE

You will be developing a logic simulator …



Many thanks to Randy Glasbergen (http://www.glasbergen.com) for permission to use his cartoon.

Contents

1 Introduction

1.1 The scenario

The aim of this project is to develop a logic simulation program in Python (Python 3, to be precise). The project involves all five major phases of the software engineering life cycle: specification, design, implementation, testing and maintenance.

The project is organised in the form of a real-life simulation. You are asked to imagine that you have joined a software development company. You have been assigned to a team of programmers who have just begun work on a contract to develop a logic simulation program. You are given the client's original requirements document and asked to produce a detailed specification for part of the system. Following this, you move on to the design stage. You are told that the program has been divided into eight functional modules and your team has been given responsibility for designing and implementing four of them. When you have completed these, you have to integrate them with the remaining four modules and then test the complete system. Finally, the client requests some changes and you are asked to implement these.

While this scenario is of course contrived, it should nevertheless give you some insight into the problems associated with large-scale software development and, hopefully, demonstrate how they can be overcome with the aid of good software engineering practice.

1.2 Working in teams

You will be forming yourselves into development teams of three people who will work together on the project. Once formed, the team must stay together for the duration of the project. You will need to register your team on the Moodle sign-up sheet before the start of the project.

Like most group efforts, some of the tasks involved can be carried out almost entirely independently, while some cannot be started until earlier tasks are complete. The responsibility for working out the order of completion rests with you, but a very rough schedule might look something like this:

Week	**Activity**
0	Form development team
1	Individual Python programming exercises
1, 2	Write specification of logic description language
1, 2	Familiarisation with supplied logic simulator code
2	Design `names`, `scanner`, `parse` and `gui` modules
2, 3	Implement `names`, `scanner`, `parse` and `gui` modules
3	Integration and final testing
4	Maintenance

There are potential problems when working in groups in this way. Three times the effort does not necessarily mean that the project will proceed three times faster, particularly if one critical component delays all the others. However, there are also many advantages. For example, it is good practice for someone other than a module's author to help write the corresponding unit tests.

1.3 Moodle

You have been enrolled on the **Part IIA Project: GF2: Software** Moodle course. There you will find: all the course documentation; some useful web links that you should use as starting points for further reading; a sign-up sheet for forming teams; a chat for in-hours support; a forum for out-of-hours support; a project calendar; and facilities for submitting reports and receiving feedback electronically.

1.4 Timetabled sessions, reports and penalties

The demonstrated project sessions are 2-4pm BST on Mondays, Wednesdays and Fridays. Please ensure that you are online at these times, to talk to each other and to seek help from demonstrators if necessary. We will be registering your attendance at a video call during each timetabled session. Outside of demonstrated hours, please post any queries to the Moodle **discussion forum**.

The following are report submission deadlines (all times are BST):

What	*Marks*	*Deadline*
1. First interim report	15 (group)	4.00pm Sunday 23 May 2021
2. Second interim report	15 (7 group, 8 individual)	9.00am Friday 4 June 2021
3. Final report	50 (individual)	4pm Thursday 10 June 2021

Three marks per day (or part thereof) are normally deducted if reports are late. Please contact Andrew Gee (`ahg13`) *in advance* if you think you may need to miss a deadline for good reason.

First interim report One submission for each group, taking the form of a single `pdf` document. It should include: introduction and description of general approach, including teamwork planning (who will do what, and when); EBNF for syntax; identification of all possible semantic errors; description of error handling; two example definition files together with diagrams showing the logic circuits they represent. See details in Section 3.

Second interim report Individual submissions for each student. The work should be complete up to the end of Section 5.5. The logic simulator should be ready for evaluation. You should submit:

1. a single zip file containing the code *you* have written (including `pytest` tests) and the alterations, if any, that you have made to the supplied code, plus a text file listing the `git` commits that *you* made (`git log --author="<your name>"`);
2. a single `pdf` document containing your example definition files, each accompanied by a diagram of the circuit represented, and a single-page user guide that should give enough information for the system to be tested using your supplied definition files. The definition files may be the same for all members of a group, but each student should write his or her own one-page user guide.

Code should be PEP 8/257 compliant or, if not, you should explain in a `README` file why you chose to deviate from the PEP recommendations (include the `README` in the zip file with your code).

Final report The code should be updated to include the maintenance work described in Section 5.6. You should submit: a single zip file containing the updated code and `pytest` tests *you* have written and your `git` commit log as above; and a single pdf document containing an individual report as detailed in Section 6. Your updated Python code, along with test definition files, must be available in a folder named `final` in the team's git repository. Please also upload to this folder a short movie (maximum 5 minutes) demonstrating the post-maintenance functionality of your simulator. Name this file `demonstration.mp4` or similar. We will use the movie in the event that we encounter any difficulties running your code on our systems.

2 Preliminary Python exercises

These exercises should be carried out individually by all students, and can be thought of as a training course you have been sent on before starting work on the system design. These exercises are not designed to cover all of Python's features, just those necessary to complete part of the project. The following instructions assume that you have a Python development environment (e.g. Anaconda) installed on your PC, and that you know how to use it. To keep all of the files associated with this preliminary work in one place, create a folder called, say, `prelim`.

1. Download the supplied code for the preliminary Python exercises (Moodle link) and unzip the files into your `prelim` folder. Open `exercise.py` in a text editor. This is a skeleton of a program that you will be completing as you work through the preliminary exercises. Read it through and make sure you understand roughly what it is supposed to do. Run the program however you normally run Python programs, for example by typing into a terminal `python exercise.py`. This should result in an error, since a command line argument is required but was not supplied. Now run it again by typing `python exercise.py example.txt`. This time the program should run correctly, displaying the output of the various `print` statements on the console. Try running the program with a different number of command line arguments, and check that you get the expected results.

2. Modify the `main` function to extract the file path from the command line arguments, print the path on the console, and then call `open_file(path)` to open the file for reading. Next, write the body of the `open_file` function, which should attempt to open the file for reading. If all goes well, `open_file` should return the file object; otherwise, it should display an informative error message before calling `sys.exit()`. This would be a good time to refresh your knowledge of Python's file I/O and exception handling facilities (Moodle links).

3. Write the body of the `get_next_character` function so that it reads the next character from the file object and returns the character to the calling routine. It should return the empty string `""` when the end of the file is reached. Again, the Moodle link on file I/O should be helpful. Next, modify the `main` function so that it calls `get_next_character` repeatedly until the end of the file is reached, printing the characters on the console one by one. If your knowledge of Python is very rusty, you might need to review Python's various loop constructs (Moodle link). Type `python exercise.py example.txt` and check that your program prints out the contents of the file correctly. If you get one character per line, instead of the characters appearing as they do in `example.txt`, then you need to tell the `print` statement not to append a newline every time it is called. You can do this my modifying `print`'s `end` parameter (Moodle link).

4. Write the body of the `get_next_non_whitespace_character` function so that it continues to read characters from the file object until it encounters a character that is not whitespace. At this point, it should return the non-whitespace character to the calling routine. It should return the empty string `""` when the end of the file is reached. You should use the `isspace` method of Python's string class (Moodle link). Next, modify the `main` function so that it resets the file pointer to the beginning of the file (Moodle link on file I/O) and then repeatedly calls `get_next_non_whitespace_character` until the end of the file is reached, printing the non-whitespace characters on the console one by one. Check that your program correctly prints out the contents of the file without spaces.

5. A number is defined as a sequence of characters that are all digits in the range 0 to 9. Write the body of the `get_next_number` function so that it searches through the file object until it encounters a number. You should use the `isdigit` method of Python's string class. The function should then return a two-element list (Moodle link) comprising the integer representation of the number, and the next non-digit character. It should return `None` for the number if none is found, and the empty string `""` for the next non-digit character if the end of the file is reached. Next, modify the `main` function so that it resets the file pointer to the beginning of the file and then repeatedly calls `get_next_number` until the end of the file is reached, printing the numbers on the console one by one. Check that your program prints out the number sequence `12`, `3`, `222`, `1` and `44`.

6. A name is defined as a sequence of characters that starts with a letter and is followed by a mixture of letters and numbers. Write the body of the `get_next_name` function so that it searches through the file object until it encounters a name. You should use the `isalpha` and `isalnum` methods of Python's string class. The function should then return a two-element list comprising the name string and the next non-alphanumeric character. It should return `None` for the name if none is found, and the empty

string "" for the next non-alphanumeric character if the end of the file is reached. Next, modify the main function so that it resets the file pointer to the beginning of the file and then repeatedly calls get_next_name until the end of the file is reached, printing the names on the console one by one. Check that your program prints out the name sequence drink, important, Ghastly, John222, exit, Terrible, hello1World and Horrid.

7. When performing lexical analysis, it is common to store names in a *name table*, so that each unique name may be referred to using an integer ID instead of its full alphanumeric string. The file mynames.py contains a skeleton class MyNames for implementing a name table. Internally, MyNames should store names in a Python list, with the list indices providing the IDs, so the first thing you should do is initialise an empty list in the class's __init__ function. Refer to the Moodle links on classes and lists as necessary. Next, write the body of the public lookup function, which should check if the given string is in the names list, and return the corresponding ID (index) if it is. If the string is not in the list, it should add the name to the list and then return the ID. Next, write the body of the public get_string function, which should return the string corresponding to the given ID, and None if the ID is not a valid index into the list.

   Import the MyNames class into exercise.py. Test MyNames by resetting the file pointer to the beginning of the file and once again repeatedly calling get_next_name, but this time look up all the names in the name table and print only those which are not bad. You will need to uncomment the two lines of code that instantiate the name table and populate it with a list of bad names. For testing purposes, be sure to use the name.get_string method to print the good names.

8. In the previous exercise, we performed some *ad hoc* tests on the MyNames class. A better approach would be to write a comprehensive suite of tests using pytest. An example is provided in the file test_mynames.py. To run the tests with verbose output, type pytest -v on the command line. If some of the tests fail, you might need to improve your implementation of the get_string method, but first you should probably read the pytest tutorial (Moodle link) so as to understand precisely what the tests are doing. When the supplied tests all pass, add some further tests for the lookup method.

9. Since software is typically read many more times than it is written, it is good practice to pay attention to style. Familiarise yourself with the PEP 8 and PEP 257 guides to Python code style and docstring conventions (Moodle links). Check that your code conforms to these standards by typing pycodestyle exercise.py and pydocstyle exercise.py, and similarly for the other files. You can generate formatted documentation for your modules by typing pydoc exercise, and similarly for the other modules.

Please keep your solutions to the above exercises as they will be useful later on.

3 Specifying the logic description language

Before starting any software design task, it is essential to have a precise specification of what the program is intended to do. Appendix A gives the client's requirements for the logic simulator. From reading this, you will see that the logic network is defined by a text file read in by the program before starting the simulation. Your main task in week 1 is to write a precise specification for the logic description language used in this text file. This specification must:

- define the syntax of the language;
- identify all the semantic constraints that apply to the language;
- define the error conditions which will be detected by the program when the definition file is read in, and state how each error condition will be reported.

Your specification for the logic description language should be designed jointly as a team and then written up as a first interim report. Note that this specification has far-reaching implications for the design of the rest of the system, so it is vitally important that the whole team participates in writing and checking it.

3.1 Syntax

The logic definition file will consist of a sequence of letters, digits and punctuation symbols. *Syntax* defines, formally, the sequence in which these may occur. In effect, the definition file can be thought of as being written in a simple language designed solely for the purpose of specifying the composition of logic circuits. Syntax specification thus involves writing a formal grammar for this language.

In this project, you are required to specify the syntax of your logic description language using the Extended Backus Naur Form (EBNF) notation. An EBNF syntax specification consists of a collection of *productions* (essentially rules), collectively called a *grammar*, that describe the formation of sentences in the language. Each production consists of a non-terminal (i.e. the name of a syntactic category) followed by a syntactic expression, separated by an equals sign. The syntactic expression defines the set of symbol strings that the non-terminal on the left denotes.

The simplest syntactic expression is just a sequence of symbols, e.g.

```
nounphrase = "the" , noun ;
```

is an EBNF syntax rule stating that a noun phrase consists of the word `the` followed by a noun. Note that terminal symbols (i.e. symbols which actually appear in the language) are written in quotes. Commas denote concatenation and each rule is terminated by a semicolon. A vertical bar is used in expressions to denote alternatives, thus

```
noun = "table" | "dog" | "cloud" ;
```

states that a noun consists of either the word `table`, the word `dog` or the word `cloud`. Wherever a symbol can appear in a syntactic expression, a nested expression can be written enclosed in parentheses. For example,

```
nounphrase = ( "the" | "a") , noun ;
```

means that a noun phrase consists of a noun preceded either by `the` or `a`. Finally, two further types of brackets are provided for nested expressions. Square brackets indicate that the enclosed expression is optional, thus for example

```
nounphrase = [ "the" | "a" ] , noun ;
```

means that a noun phrase consists of either a noun on its own or a noun preceded by `the` or `a`. Curly brackets indicate that the enclosed expression is repeated zero or more times. For example,

```
identifier = letter , { letter | digit } ;
```

means that an identifier consists of a single letter followed by zero or more letters and digits.

To summarise, an EBNF grammar consists of a set of rules as defined by the following EBNF grammar!

```
EBNFgrammar    = { rule } ;

rule           = nonterminal , "=" , expression , ";" ;

expression     = term , { "|" , term } ;

term           = factor , { "," , factor } ;

factor         = nonterminal | terminal | "(" , expression , ")" |
                 "[" , expression , "]" | "{" , expression , "}" ;
```

```
nonterminal    = letter , { letter | digit } ;

terminal       = "'" , character , { character } , "'" |
                 '"' , character , { character } , '"' ;
```

where

"|" denotes alternatives, e.g. a|b means a or b

"( )" denote factors, e.g. (a|b),c means ac or bc

"[ ]" denote options, e.g. a,[b],c means ac or abc

and "{ }" denote repetitions, e.g. a,{b,c} means a or abc or abcbc etc.

You will need to provide a mechanism for comments to be included in the logic definition file. However, you should design the scanner rather than the parser to remove the comments from the input file, so the comment syntax does not need to be included in the formal grammar definition. Instead, it can be specified in general terms and a suitable filter implemented in the scanner.

3.2 Semantics

Logic definition files which obey the syntactic rules but nevertheless do not describe meaningful logic circuits are said to contain *semantic* errors. In this project, we will not concern ourselves with formal methods for specifying semantics. Instead, semantics are to be specified informally using plain English. For example, suppose that you decide to specify a connection in your logic description language using the following syntax rule:

```
connection = signalname , "->" , signalname ;
```

Such a rule might be associated with the following semantic constraint:

> The signal name to the left of the "->" symbol must be the name of a device output and the signal name to the right must be a device input.

3.3 Errors

There are two general classes of error which may occur in a definition file: syntax errors and semantic errors. A syntax error occurs when the sequence of symbols in the definition file fails to follow the prescribed EBNF syntax rules. A semantic error occurs when the syntax is correct but the symbol string is meaningless. For example, the connection string

```
G1 -> G2
```

would raise a semantic error if both G1 and G2 were outputs.

In your first interim report, you should include:

- A general statement about how errors will be reported.
- A statement about how syntax errors will be handled.
- Identification of all possible semantic errors and a statement about how they will be detected and reported.

Note that the time spent on a detailed specification of error handling at this stage is rarely wasted, as such a specification can be used directly when writing the code.

3.4 General guidelines

In designing your logic description language, you should bear in mind the following guidelines.

- Definition files should be readable. For example,

  ```
  G1 G2 NAND 2 SW1 SW2 SWITCH/
  G1.I1 SW1 G2.I2 SW2 G2.I1 G1 G1.I2 G2/
  G1 G2
  ```

  is a functional definition of the example network in Appendix A, but it is not readable. Use English keywords and punctuation, so that the meaning is clear to a human reader as well as a machine.

- Make sure that your syntax can be processed by a top-down, single lookahead parser (if you don't know what this means, see Section 5.3).

- Consider the effects of syntax errors when designing your language. Include punctuation so that a parser can resume proper operation as soon as possible after locating a syntax error. For example, if connections as specified in the example of Section 3.2 had to be terminated by semicolons, then following a syntax error in a connection expression, the parser could easily skip text until the next semicolon and then resume normal operation.

- Your syntax must be free-format, i.e. the users of the simulator must be able to lay out the text as they wish using spaces and line breaks freely. In particular, the end of a line must have no syntactic significance (apart from possibly terminating a comment).

4 Collaborative coding and version control

It is now time to obtain a copy of the supplied source code for the logic simulator (Moodle link), the structure of which is described in detail in Appendix B. One member of each team should initialize a `git` repository with this code, and then grant access to the other two members.

You may use any `git` platform you like. In Part IA, students were encouraged to use GitLab, but feel free to migrate to BitBucket, GitHub or any other service of your choice. In all cases, though, please make sure that your repository is *strictly private*. Once your repository is correctly initialized, please grant Andrew Gee (`ahg13`) and Tim Love (`tl136`) read access, so that they can help you when you get stuck and download your final code for assessment.

We assume that students are familiar with collaborative coding and version control using `git`, most likely from the Part IA flood warning exercise. However, we do understand that some of you may have migrated to Engineering after Part I. If `git` is new to you, please seek help from your teammates, read **A Simple Guide to Git** (Moodle link) and explore other online tutorials.

As well as collaborative coding and version control, many of the online `git` platforms offer other services, such as issue trackers and automatic unit testing (continuous integration). Please take full advantage of these services, and tell us how you used them in your final report. You might also choose to use more generic "brainstorming" platforms like Trello and MindMeister.

5 Implementation guidelines

5.1 The names module

The purpose of the `Names` class is to map variable names and string names to unique integers. The former is useful for handling error codes returned by functions, by assigning unique integer error codes to meaningful variable names. This part of the class is supplied fully implemented in `names.py`. In contrast, you will need to write the methods for translating name strings to integer name IDs, though you should be able to recycle some of your code from the preliminary Python exercises. You will, however, need to modify the `lookup` member function to accept a list of name strings and return a corresponding list of name IDs. You will also need to write the `query` function, which returns the name ID for a single name string, or `None` if the name string is not found in the name table (unlike `lookup`, which adds new names to the name table). Then you should design some `pytest` tests to thoroughly test the class.

5.2 The scanner module

This module contains two classes. The `Symbol` class encapsulates a single *symbol*, which might be, for example, a keyword, a name, a number or a punctuation symbol. The distinction between keywords and names is more convenient than necessary, making it easier to check whether any given name is a reserved keyword or not. A partial implementation of the `Symbol` class is supplied, comprising the symbol's type and integer ID, though you might wish to extend the class to include, say, the symbol's line number and position on the line.

The purpose of the `Scanner` class is to translate the sequence of characters in the definition file into a sequence of symbols for consumption by the parser. The `Scanner` class also takes care of skipping over comments and irrelevant formatting characters such as spaces and line breaks. Suppose your logic description syntax utilises the following symbols:

- User-defined names
- Keywords `DEVICES CONNECT MONITOR END`
- Numbers
- Punctuation symbols `,` `;` `=`

Then the `Scanner` class's initialiser method might look something like this.

```
def __init__(self, path, names):
    """"Open specified file and initialise reserved words and IDs."""

    self.names = names
    self.symbol_type_list = [self.COMMA, self.SEMICOLON, self.EQUALS,
        self.KEYWORD, self.NUMBER, self.NAME, self.EOF] = range(7)
    self.keywords_list = ["DEVICES", "CONNECT", "MONITOR", "END"]
    [self.DEVICES_ID, self.CONNECT_ID, self.MONITOR_ID,
        self.END_ID] = self.names.lookup(self.keywords_list)
    self.current_character = ""
```

The initialiser is passed an instance of the `Names` class and the path to the logic definition file. It opens the definition file (code not shown), initialises a list of symbol types (including `EOF` for the end of the file) and populates the name table with the keywords. `names.lookup` is similar to the `lookup` function in the preliminary exercises, except that it accepts a list of strings and returns a list of IDs. `init` also initialises a variable, `current_character`, to hold the last character read from the definition file.

The most important member function is `get_symbol`, which is called repeatedly by the parser to return successive symbols from the logic definition file. An implementation of `get_symbol` might look something like this.

```
def get_symbol(self):
    """Translate the next sequence of characters into a symbol."""
```

```
symbol = Symbol()
self.skip_spaces()  # current character now not whitespace

if self.current_character.isalpha():  # name
    name_string = self.get_name()
    if name_string in self.keywords_list:
        symbol.type = self.KEYWORD
    else:
        symbol.type = self.NAME
    [symbol.id] = self.names.lookup([name_string])

elif self.current_character.isdigit():  # number
    symbol.id = self.get_number()
    symbol.type = self.NUMBER

elif self.current_character == "=":  # punctuation
    symbol.type = self.EQUALS
    self.advance()

elif self.current_character == ",":
# etc for other punctuation

elif self.current_character == "":  # end of file
    symbol.type = self.EOF

else:  # not a valid character
    self.advance()

return symbol
```

Here, the member function `get_name` is similar to `get_next_name` in the preliminary exercises, except that it now assumes the current character is a letter, returns only the name string and places the next non-alphanumeric character in `current_character`. Similarly, `get_number` assumes the current character is a digit, returns the integer number and places the next non-digit character in `current_character`. `advance` reads the next character from the definition file and places it in `current_character`, while `skip_spaces` calls `advance` as necessary until `current_character` is not whitespace.

You will need to equip the `Scanner` class with a method to print out the current input line along with a marker on the following line to show precisely where an error occurred: see the section on error handling below for further details. Remember to write `pytest` tests for this and all other aspects of the class.

5.3 The parse module

The interface to the `Parser` class should consist of a single function

```
def parse_network(self):
```

which returns a boolean value indicating whether parsing was successful or not. The function analyses the syntactic and semantic correctness of the symbol sequence returned to it by repeated calls to the `getsymbol` function in the `Scanner` class, and builds the corresponding logic network using the routines in the supplied `Devices` and `Network` classes. Your development of this class should take place in the following stages.

- Design the `parse_network` function initially so that it only analyses the definition file (i.e. it makes no calls to the `Network` or `Devices` classes).

- Write tests using `pytest` and prepare suitable test definition files. Note that you will only need the `Names` and `Scanner` classes to do this; they would need to be complete and tested at this point.
- Finally, insert the appropriate calls to the functions provided in the `Network` and `Devices` classes, and add further `pytest` tests.

The following sections suggest how the above might be done.

Top down parsing

Each EBNF syntax rule in your logic description language specification can be translated directly into a function which parses that rule. This translation is performed as follows.

1. Non-terminal symbols on the RHS of a rule are translated into a call to the function which parses that rule.
2. Terminal symbols on the RHS of a rule are translated into a check that the current input symbol is one of the required terminal symbols, followed by a call to `scanner.getsymbol` to get the next symbol.
3. Syntactic expressions of the form

   ```
   [ "x" , y ]
   ```

   are translated directly into

   ```
   if current_symbol == xsym:
       symbol = scanner.getsymbol()
       y()
   ```

4. Syntactic expressions of the form

   ```
   { "x" , y }
   ```

   are translated into

   ```
   while current_symbol == xsym:
       symbol = scanner.getsymbol()
       y()
   ```

5. Syntactic expressions of the form

   ```
   ( "x" , y ) | ( "u" , v )
   ```

   are translated into

   ```
   if current_symbol == xsym:
       symbol = scanner.getsymbol()
       y()
   elif current_symbol == usym:
       symbol = scanner.getsymbol()
       v()
   else:
       error()
   ```

Notice in the above the need for a terminal symbol at the start of each syntactic expression where a choice is possible. Technically, a grammar which allows parsing by this method is called LL(1), meaning left to right with one lookahead symbol.

As an example, suppose that your syntax rules expect connections to be written with the following syntax:

```
connectlist = "CONNECT" , connection , { "," , connection } , ";" ;
connection = signame , "=" , signame ;
```

A typical connection list might be

```
CONNECT A  = B.I1,
        SW = B.I2,
        B  = A.I1;
```

Then a function to parse a single connection would have the form

```
def connection(self):
    self.signame()
    if self.symbol.type == self.scanner.EQUALS:
        self.symbol = self.scanner.get_symbol()
        self.signame()
    else:
        self.error()
```

and for the connection list itself

```
def connection_list(self):
    if (self.symbol.type == self.scanner.KEYWORD and
        self.symbol.id == self.scanner.CONNECT_ID):
        self.symbol = self.scanner.get_symbol()
        self.connection()
        while self.symbol.type == self.scanner.COMMA:
            self.symbol = self.scanner.get_symbol()
            self.connection()
        if self.symbol.type == self.scanner.SEMICOLON:
            self.symbol = self.scanner.get_symbol()
        else:
            self.error()
    else:
        self.error()
```

Error handling

In the above, the points at which errors are detected are just marked by a call to a parameter-less function called `error`. In practice, rather more than this is needed:

- The nature of the error must be reported.
- Symbols must be skipped until a suitable point to resume parsing is reached.
- A count should be kept of the total number of errors detected.

Error reporting is fairly straightforward. As an example, you could call an error function with a number denoting an error message to print:

```
def display_error(self, error_type):
    self.error_count += 1
    if error_type == self.NO_NUMBER:
        print("Expected a number")
    elif error_type == self.NO_EQUALS:
        print("Expected an equals sign")
    etc...
```

Additionally, you are required to add a function to your `Scanner` class so that it prints out the current input line along with a marker on the following line to show precisely where the error occurred, e.g.

```
CONNECT   A B.I1
            ^
***Error: Expected an equals sign
```

Error recovery is more difficult. The simplest scheme is to pass the value of a *stopping symbol* to the `error` function and add code to skip input symbols until the given stopping symbol has been found:

```
def display_error(self, error_type, stopping_symbol):
    ...
    while (self.symbol.type != stopping_symbol and
           self.symbol.type != self.scanner.EOF):
        self.symbol = self.scanner.get_symbol()
}
```

A more sophisticated scheme is to use sets of stopping symbols rather than a single stopping symbol. The idea is that each analysis function passes its own stopping symbol set to any function it calls. Thus, each analysis function can add its own stopping symbols to those given to it by its caller. This ensures that the error routine will stop skipping at the earliest point at which normal parsing can be resumed, and also that missing stopping symbols due to further errors in the definition file do not necessarily result in the rest of the file being skipped.

Semantic analysis and network construction

Once the `Parser` class has been thoroughly tested for syntax handling, statements to perform semantic analysis can be added. You will need to call functions in the `Devices` and `Network` classes. Here, an illustration of the approach is given. The example `connection` syntax described earlier is intended to specify a connection between a device input and a device output. Supposing that the `signame` function returns IDs for the device name and the input/output port name, then the `connection` function can now be extended as follows:

```
def connection(self):
    [in_device_id, in_port_id] = self.signame()
    if self.symbol.type == self.scanner.EQUALS:
        self.symbol = self.scanner.get_symbol()
        [out_device_id, out_port_id] = self.signame()
    else:
        self.error(...)
    if self.error_count == 0:
        error_type = self.network.make_connection(
            in_device_id, in_port_id, out_device_id, out_port_id)
        if error_type != self.network.NO_ERROR:
            self.error(...)
```

Here the required connection is made by a call to the `make_connection` function in the `Network` class. Note that all calls such as these should only be made while the total number of errors is zero. In other words, once the first error occurs, all subsequent attempts to construct the network should be abandoned.

5.4 The GUI module

If you run `logsim.py` with the `-c` flag, once the `Scanner` and `Parser` classes have completed their job, control passes to the `command_interface` routine in the `UserInterface` class (see the supplied file `userint.py`). At this point, the simulation may be run, after which the `display_signals` function from the `Monitors` class is called to produce a text-based display of the logic signals being monitored. While this is adequate for testing purposes, the client also requires a graphical user interface (GUI).

You should design and implement your GUI using the wxPython toolkit. This is a publicly available suite of Python classes with many attractions:

- It is powerful and yet easy to use: sophisticated GUIs can be created with just a few lines of wxPython code.
- It is well documented and well supported, see the links on the Moodle course page.
- It is also available in C++ (wxWidgets) and is portable: whether written in Python or C++, your GUI code should run, without modification, on a variety of platforms, including Windows, Linux and macOS.
- It is free: the terms of the wxPython license allow even commercial use free of charge.

Running `logsim.py` without the `-c` flag launches the skeleton GUI in the module `gui.py`. The supplied code includes examples of how to create things (widgets) like pull-down menus, buttons, text entry boxes, pop-up dialogues and graphics drawing areas. The code is far from exhaustive but should help get you started. Note how various user actions (pressing a button, entering some text, clicking the mouse in the drawing area) generate *events* that cause *event handler* functions to be executed. This is a common programming model for GUIs: you might want to do some background reading if you have not come across it before.

The graphics drawing area (`MyGLCanvas`) in `gui.py` makes use of the OpenGL graphics language, which is portable and often accelerated by dedicated graphics hardware. OpenGL supports full 3D graphical rendering, including viewing and modelling, illumination, reflection, texture mapping and transparency, and is certainly overkill for drawing simple 2D traces. The use of OpenGL in this project is largely pedagogical, a convenient opportunity for students to gain a little experience with this powerful and popular library. That said, an exceptionally blingy GUI might actually require some 3D functionality.

In the supplied code, the `MyGLCanvas.render` function draws an artificial signal trace to illustrate the operation of the OpenGL functions `glBegin, glVertex2f, glEnd,` etc. You will not find these functions described in the wxPython documentation. Instead, you will need to read about them in the system manual pages (e.g. type `man glBegin` on the command line) or in the *OpenGL Programming Guide* (Moodle link). The `MyGLCanvas.render_text` function illustrates OpenGL Utility Toolkit (GLUT) instructions for displaying text in the graphics drawing area, and should be self-explanatory.

Your GUI should provide the following functionality currently in the `UserInterface` class, and perhaps go beyond this at your discretion:

```
r N      - run the simulation for N cycles
c N      - continue simulation for N cycles
s X N    - set switch X to N (0 or 1)
m X      - set a monitor on signal X
z X      - zap the monitor on signal X
q        - quit the simulation
```

You should refer to the various `UserInterface` and `Monitors` member functions for examples of how to run the network, add monitors, access monitored signal traces and so on: a lot of this code can be cut and pasted into your new `gui.py` module. There are no hard rules for how the GUI should look and feel, you will need to think about the best combination of wxPython controls and displays. At one end of the spectrum, you could have a single text entry box, like the one already provided in `gui.py`, into which the user types the above commands. If you follow this route, you will only need to worry about displaying the monitor signals in the OpenGL drawing area: the rest of the functionality can be achieved by cutting and pasting from the `UserInterface` class. At the other end of the spectrum, everything could be mouse driven. You could even supplement `logsim`'s command line argument with an option to choose the logic definition file in a `wx.FileDialog`. Some paper and pencil time, before you start coding, would be a good idea. Think particularly carefully about how the display routines will cope with the drawing area being resized (when the user resizes the window): this may be problematic if the drawing area is small and there are a large number of monitor points and/or long signal traces.

We recommend that only the `gui` module should make use of wxPython and OpenGL (although the main program in `logsim.py` creates the `wx.App` object and launches the main event loop). Such modularisation makes it easier to change the logic simulator to use different GUI and graphics libraries, if required in the future. Like all the other classes, you should test the `Gui` class as thoroughly as possible before attempting to use it in the integrated system, though formal unit testing of GUIs is impractical. Note that it will be difficult to test some routines until the `Parser` and `Scanner` classes are available.

5.5 System integration

By this stage in the project, you should have fully tested all the modules needed to build the logic simulation software, with the exception of the `gui` module that requires interactive testing in the integrated system. Integration testing might proceed as follows:

1. Prepare a set of definition files to exercise as many of the system's degrees of freedom as possible.
2. Run `logsim.py` without the `-c` flag. It is good practice for the tester to be somebody who was not the author of the component under test. Testers should try to break the software by fair means or foul, typically by operating it in ways not intended by the author.
3. When you spot errors, fix them and also, where possible, add extra `pytest` tests that would have detected these errors automatically.

Keep an orderly record of your test results, including screenshots of the GUI for the interactive tests.

5.6 Maintenance

Having been shown the working logic simulator, the client has decided that they would like some modifications made to the system. These will be detailed in a memo posted on the Moodle course page at 9am BST on Friday 4 June 2021 and must be complete by the time your final report is submitted (4pm BST on Thursday 10 June 2021). By withholding the required modifications until this time, we are essentially forcing you to make the necessary changes quickly. This is typical of real world programming scenarios. You should plan for this sort of maintenance by structuring your code sensibly, with plenty of modularisation.

6 Final report

You must leave the master `git` repository intact and also create a new folder called `final`, into which you should place your final Python code, a demonstration movie and all of your test definition files. We will be using this, along with the information provided in Appendices C and D of your final reports, to test the new functionality added in the maintenance phase. It is important that your code just works “out of the box” the first time we run it: imagine us as the client finally receiving the software we have paid for.

Additionally, you should submit: a single zip file containing the updated code and `pytest` tests *you* have written and the alterations, if any, that you have made to the supplied code, plus a text file listing the `git` commits that *you* made (`git log --author="<your name>"`); and a single pdf document containing an individual report which should be structured as follows:

- Title page with your name, team number, College and userid.
- Introduction.
- Description of the function of the logic simulator and software structure.
- Commentary on the approach taken to teamwork: did things progress as anticipated?
- Commentary on the approach taken to remote collaboration: which online tools did you use?
- Description of the software written and/or modified by you.
- Description of the test procedures adopted.
- Conclusions and recommendations for improvements.

Appendices:

A. Example definition files, diagrams of the circuits they represent and copies of test results obtained, including new features added during the maintenance phase. This appendix may be shared amongst team members.

B. Specification of the logic description language, including any changes made during the maintenance phase. This appendix may be shared amongst team members.

C. Single-page user guide for the software, including new features added in the maintenance phase. This appendix must be written by you.

D. A brief description of the files in your `final` team folder. This appendix may be shared amongst team members.

The length of this report should not exceed 10 pages of A4 when typeset in a 10-point font (excluding title page and appendices). It should be well presented, using a professional typesetting system like LaTeX or (if you must) LibreOffice/Word. Handwritten submissions are not acceptable.

A Client’s requirements for the logic simulator

A.1 Function

The purpose of the logic simulator shall be to enable the operation of both combinatorial and clocked logic circuits to be studied by simulation on a computer prior to their implementation in hardware. The program will operate in two phases. In the first phase, it will read in a text file which defines the logic elements (gates, bistables etc.), the connections between them, the generators needed for inputs (switches and clocks) and the signals to monitor. In the second phase, the logic network defined in the first phase will be executed under user control. The principal user functions in this second phase will be to run the network for a given number of simulation cycles, to change the states of switch inputs, and to add and remove monitor points on signals.

A.2 Devices

The following list of devices will be provided in the initial implementation of the logic simulator.

- `CLOCK` — 1 output, 0 inputs
  Function: Output changes state every n simulation cycles, where n is specified in the definition file.
- `SWITCH` — 1 output, 0 inputs
  Function: Output is either 1 or 0. Initial value is specified in the definition file but can be changed by a user command during the simulation.
- `AND NAND OR NOR` — 1 output, 1–16 inputs
  Input Names: I1 I2 I3 .... etc.
  Function: Logic gates with the usual boolean function.
- `DTYPE` — 2 outputs, 4 inputs
  Input Names: DATA CLK SET CLEAR
  Output Names: Q QBAR
  Function: QBAR is always the inverse of Q. Logic 1 on the SET input forces Q high, logic 1 on the CLEAR input forces Q low. Otherwise, input to DATA is transferred to Q on rising edge of the CLK input.
- `XOR` — 1 output, 2 inputs
  Input Names: I1 I2
  Function: Output is high when I1 is high or I2 is high, but not both.

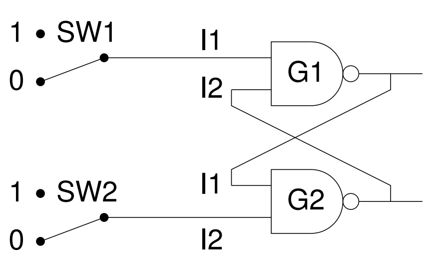
A.3 The definition file

The definition file will be a text file consisting of letters, digits and punctuation symbols read in by the program to:

- define the list of devices;
- give a user-defined name to each such device;
- specify any parameters necessary to configure the device (initial switch setting, clock repetition period, number of gate inputs etc.);
- define the connections between devices;
- specify the initial set of output signals to monitor in the subsequent simulation.

If `X` is the name of a device with input `I`, then this input will be referred to by the notation `X.I`. If the device has only one output, then this output will be referred to simply as `X`, otherwise multiple outputs will also be named and referred to using the same dot notation (e.g. `X.Q`, `X.QBAR`).

An example might be to provide a notation for the following description of a circuit.




```
G1 and G2 are NAND gates with 2 inputs
SW1 and SW2 are SWITCHes, initially with 0 outputs
SW1 is connected to G1.I1
SW2 is connected to G2.I2
G1 is connected to G2.I1
G2 is connected to G1.I2
The initial monitor points are G1 and G2
```

A.4 Text-based user interface

Once the logic network has been set up, the program should prompt the user to enter a command. These commands should consist of a single letter followed by one or more arguments as appropriate. The set of commands provided should include:

1. Run (`r N`)

   Run the simulator for `N` cycles and display the waveforms at the current monitor points. Assume a cold start-up of the circuit, so randomise the state of `DTYPE` latches and clocks before the first simulation cycle. Clear any existing signal traces.

2. Continue (`c N`)

   Continue running the simulator (e.g. after changing a switch) for `N` cycles and display the waveforms at the current monitor points.

3. Set switch (`s X N`)

   Set switch `X` to `N` (0 or 1).

4. Set monitor point (`m X`)

   Add the output signal `X` to the current list of monitor points.

5. Zap monitor point (`z X`)

   Remove the monitor point `X`.

6. Help (h)

   Print a list of available commands on the terminal.

7. Quit (q)

   Quit the simulation.

There should also be a text-based display of the signals recorded at the various monitor points.

A.5 Graphical user interface

A subsequent version of the logic simulator should provide a graphical user interface, offering at least the functionality described in 1–5 above. There should also be a graphical display of the signals recorded at the various monitor points. The text-based interface should be preserved as a user-selectable option.

A.6 System constraints

In designing the logic simulator, the following list of constraints must be adhered to.

1. The syntax of the definition file should be simple to understand and unambiguous. Errors in the definition file should be reported fully and in such a way that the user can easily locate each error. A count of the total number of errors should be displayed.

2. Full error checking should be applied to user command inputs. The system must be robust and easy to use.

3. There shall be no limit on the number of devices in a network, or on the number of monitor points, except for that implied by the available memory of the computer.

B Supplied software

This appendix describes the overall system design and gives details of the supplied Python modules.

B.1 Software components

The following table shows which pieces of software are supplied and which are to be designed and implemented by you.

module	**supplied?**
`names`	skeleton only
`scanner`	skeleton only
`parse`	skeleton only
`devices`	yes
`network`	yes
`monitors`	yes
`userint`	yes
`gui`	demonstration only
main program	
`logsim`	yes

There is a supplied implementation of the `gui` module, though this is just to get you started and will require substantial modification. You may also need to modify other supplied modules in the maintenance phase of the project.

B.2 Operation overview

In the first phase of the program, the parser is called to read the definition file and build the corresponding logic circuit. The parser does not read the definition file directly, instead it does this via the `Scanner` class which breaks the sequence of characters into symbols, returning them one by one to the parser. As each syntactic construction is parsed, the parser calls methods in the `Devices` and `Network` classes to insert devices into the network and make the required connections between them. It also calls a method in the `Monitors` class to set monitor points on output signals.

Provided that no errors were detected in the first phase, the second phase of program execution is commenced. This makes use of a command interpreter in the `UserInterface` class, which repeatedly reads commands typed in by the user and executes them by making appropriate calls to methods in the `Devices`, `Network` and `Monitors` classes. While this is adequate for testing the `parse` and `scanner` modules, you must eventually offer a more sophisticated graphical user interface, implemented in the `gui` module, while retaining the option for the user to select the existing text-based interface should they so wish.

Finally note that throughout the system, names are represented internally by a unique integer ID. The mapping between the name string and the name ID is managed by the `Names` class which is used by most of the other classes. The `Names` class also manages the mapping of error names to unique integers.

The following notes give more detail on each of the supplied modules.

B.3 The devices module

This module contains two classes. The `Device` class encapsulates a single device, recording: the device's name ID; a dictionary of the device's inputs and the outputs they are connected to; a dictionary of the device's outputs and what their signal levels are; what kind of device it is; and ancillary device-specific state variables.

The `Devices` class contains routines for creating, configuring and querying devices. All devices defined in the logic definition file should be instantiated by calls to the function `make_device`. This takes parameters which define the name of the device, the kind of device and an optional qualifier. `make_device` calls device-specific functions which in turn call `add_device`, `add_input` and `add_output` to build each device. The qualifier is an integer number interpreted according to the kind of device as follows:

`SWITCH` — qualifier defines initial state of switch, 0 = low, 1 = high

`CLOCK` — qualifier defines number of simulation cycles in half a clock period

`AND, NAND, OR, NOR` gates — qualifier defines number of inputs (16 maximum)

For other device kinds, `make_device` returns an error if a qualifier is supplied.

The `Devices` class contains also a number of helper functions for mapping between device IDs and objects, and also between signal names and device/port IDs. Unique error codes are assigned to device-related errors using the `unique_error_codes` method in the `Names` class. Signal levels are defined as `LOW`, `HIGH`, `RISING`, `FALLING` or `BLANK`. `RISING` and `FALLING` are required because some logic devices are edge triggered. `BLANK` is used by the `Monitors` class when monitors are added part way through a simulation, in which case partially blank signal traces will need to be displayed. Finally, there are functions to set the state of a switch, and to randomise the state of `DTYPE` latches and clocks on a cold start-up.

B.4 The network module

This module contains the `Network` class, which manages connections between devices and the subsequent execution of the logic circuit. A key member function is `make_connection`, which is called to establish a connection between the output of one device and the input of another. This involves careful error checking, for example to check that an input is not being connected to another input, or that an input is not being connected to a second output. Unique network-specific error codes are assigned through the `unique_error_codes` method in the `Names` class. Once all the connections have been made, the `check_network` function may be called to verify that all inputs are connected to an output.

After the network has been built, it can be executed for one simulation cycle by calling the function `execute_network`. This first calls `update_clocks` to set clock edges to `RISING` or `FALLING` as necessary. It is important to do this before executing any edge-triggered `DTYPE` devices. Next, every device in the circuit is executed by calling `execute_zzz`, where `zzz` is the device kind. These functions define how each device kind operates. Essentially, they work by examining the signal levels at each input and then setting the outputs accordingly. Output signal levels are set by the `update_signal` function. This function ensures that signal transitions between low and high always involve moving the signal through the falling or rising states. In addition, `update_signal` sets a boolean flag called `steady_state` to false whenever a signal actually changes state. `execute_network` continues to call the various `execute_zzz` functions for all the devices in the network, until one complete pass through the device list leaves the `steady_state` flag true. At this point, the network has settled and the simulation cycle is complete.

The `Network` class also contains helper functions for querying signal levels at inputs and outputs, and discovering which output is connected to any particular input. Finally, there is a helper function to invert signals between `HIGH` and `LOW`.

B.5 The monitors module

This module contains the `Monitors` class, whose purpose is to record and display (on a text console) the signal levels at the designated monitor points for each simulation cycle. The signal levels are stored in a dictionary that maps a tuple, comprising the device ID and the output ID, to a list of signals. Since this is an *ordered* dictionary, the monitors can be displayed in the same order as they are defined in the logic definition file. The `record_signals` function is called at every simulation cycle to append the latest set of signals to the lists in the monitor dictionary. `record_signals` in turn calls a helper function to obtain individual, current signals from the `Network` class. Monitor points can be added and removed via calls to `make_monitor` and `remove_monitor` respectively, while all monitor traces are cleared by `reset_monitors`. Finally, `display_signals` draws the complete set of monitor traces on the text console, with a helper function to calculate the width of the margin required for each monitor's name.

B.6 The userint module

This module provides the `UserInterface` class, which handles the text-based user interface. There is only one outward-facing member function, `command_interface`, which is called by the main program after the logic definition file has been successfully parsed. The body of the `command_interface` function consists of a `while` loop which reads a command line from the user's terminal and then calls one of a set of command functions dependent on the first letter in the command line.

Each command function implements one of the commands specified in the client's requirements list (see Appendix A.4). The command functions pick up any necessary arguments from the command line using lexical analysis helper functions, and then call the relevant routines in the `Devices`, `Network` and `Monitors` classes. You will need to replicate much of this behaviour when you write the `Gui` class.

B.7 The GUI module

The supplied module `gui.py` shows how to create a wxPython graphical application window with some controls and an OpenGL drawing area. You can experiment with this skeleton GUI by running `logsim.py` without the `-c` flag. In this way, development work can proceed before the `Names`, `Parser` and `Scanner` classes are available. A complete understanding of the `gui` module requires extensive reference to the wxPython and OpenGL documentation, but what follows provides a broad overview.

The `main` function in `logsim.py` creates `wx.App` and `Gui` objects. The call to `gui.Show` causes the GUI to be displayed. The call to `app.MainLoop()` transfers control to the wxPython main loop: from now on, program execution is governed by events and event handlers.

The `Gui` class is defined in `gui.py`. It is derived from the wxPython `wx.Frame` class, which provides a resizable and movable window. The `__init__` function of the `Gui` class receives a number of parameters which it currently ignores, though you will find them useful when you come to develop the GUI into its final, functional form. `__init__` goes on to create some illustrative user interface controls and an OpenGL drawing area. Event handler functions are then bound to the various controls: the event handler is executed automatically whenever the control is activated. So, for example, whenever the 'run' button is pressed, a `wx.EVT_BUTTON` event is emitted. We bind this event to the `on_run_button` event handler, which consequently gets called whenever the button is pressed. The various controls and the drawing area are placed in a nested hierarchy of *sizers*, in this example we use the `wx.BoxSizer`. Sizers govern the layout of the objects placed inside them, and how the layout changes when the parent window is resized. For more details about this important topic, refer to the *wxWidgets Tutorials* (Moodle link).

The OpenGL drawing area is a `MyGLCanvas` object, derived from the wxPython `wxcanvas.GLCanvas` class. In the class's `__init__` function, you can see some of the events associated with OpenGL drawing areas, and how they are bound to event handler functions. There are currently handlers for when the canvas is resized, when it needs redrawing (e.g. when it is exposed or partially obscured by another window) and when the user performs mouse operations inside it. These functions contain a mixture of wxPython and OpenGL code. For example, when the window is resized it is necessary to reinitialise the OpenGL drawing context (via the `init_gl` function) with calls to `GL.glViewport` and `GL.glOrtho`: for more details, see the system manual pages and the *OpenGL Programming Guide* (Moodle link). The `render` function deals with displaying the contents of the OpenGL drawing area. It is called by the `on_paint` event handler and whenever else we want to redraw the display, for example in the `Gui.on_run_button` event handler. Its current contents are for illustrative purposes only: it writes some text on the display and draws a dummy signal waveform.

Other parts of `gui.py` should be fairly self-explanatory, but you will need to refer to the wxPython and OpenGL documentation. Note how easy wxPython is once you get the hang of it. For example, the 'About' item in the 'File' menu launches a pop-up dialog which displays a message and freezes the application until the 'OK' button is pressed: all this in just two lines of code in the `Gui.on_menu` event handler.

Andrew Gee and Mojisola Agboola
April 2021