ENGINEERING TRIPOS PART II A

GF2 SOFTWARE P2

INTERIM REPORT 1 - INTRODUCTION, GENERAL APPROACH, SYNTAX AND ERROR HANDLING

Name: Group 7: Daniel MacKinnon, Léa Gansser-Potts & Robbie Sewell

Colleges: Queens', Queens' & Fitzwilliam

Report due: 18th May 2019 at 4pm

1 Introduction

The aim of this project is to emulate a professional software development environment in which our team, made out of three software developers, is tasked with designing and implementing an efficient logic circuit simulator package. This document outlines the structure put in place and the attitude adopted by the team to ensure that the challenges faced are overcome and a high quality product is delivered.

2 General Approach

The general approach to the task is to keep the solutions simple and elegant.For instance, the grammar developed remains readable and as close to English as possible, all the while having a specific structure that allows for the parser's functions to be straightforward and robust. We aimed to complete a first iteration of the code within the first two weeks of the project, allowing time to build on this first draft as well as developing a intuitive, simple and straight-forward user interface, with features such as a 'live parser' that would tell the user if they are making any syntactic or semantic errors as they input their circuit.

The user guide and the error handling mechanisms are being continuously added to as the software is developed in order to ensure that all bases have been covered, and no details missed.

As far as possible, the team has endeavoured to make collective and conscientious decisions at the start of the project regarding the language grammar and the timings that will allow for a simple implementation.

A modular approach has been adopted with the code, and specifically, the different functions in the `parser.py` module have been written by different team members. This was done in order for different solutions to similar problems to be found, and for the team to assess which solution is the most robust, before integrating and homogenising the software. Similarly, the GUI module will be done in a modular way, assigning the different features to individual team members, so that different approaches are considered and the most appropriate one followed. This modular approach has been adopted to

easily spot where bugs in the program are, and for flexibility in the maintenance stage.

2.1 Teamwork planning

The project has been divided into tasks, all of which have been given an 'owner' and a 'deadline'. The 'owner' is responsible for understanding all the elements of the task and ensuring that it is delivered within the 'deadline'. That being said, the task 'owner' is not the only contributor to that task. A detail of the project schedule is given below:

SUB TASK	OWNER	DEADLINE
EBNF language	Dan	16/05/19
Teamwork planning	Léa	16/05/19
Error handling (syntatic and semantic)	Robbie	16/05/19
Example circuit definition files x 2	Léa	18/05/19
Interim report 1	**Léa**	**18/05/19 at 4pm**
`scanner.py`	Robbie	18/05/19
`names.py`	Robbie	18/05/19
Overall `parser.py`	Robbie	20/05/19
`parser.py devices` function	Robbie	20/05/19
`parser.py init` function	Léa	20/05/19
`parser.py connections` function	Dan	20/05/19
`parser.py monitor` function	Robbie	20/05/19
Integration	Joint	20/05/19
GUI and pytests planning meeting		Lab session 6 (20/05/19)
GUI	Robbie and Dan	28/05/19
pytests	Léa	28/05/19
User guide	Dan	30/05/19
Interim report 2	**Individual**	**30/05/19 at 11am**
Maintenance	tbc	05/06/19
Final report	**Individual**	**05/06/19 at 4pm**

Table 1: Work schedule

Full planning of the GUI module and the pytest have not been undertaken at this point, the schedule will be completed after the planning meeting scheduled for the 20/05/19.

Furthermore *git* has been used to track the contribution of the different teams members, but only the software has been included on the shared repository, which can be found at `https://github.com/rsewell97/GF2-Software`.

3 EBNF for syntax

The attitude adopted was to keep the `.txt` file as readable and flexible as possible, while keeping the language straightforward to interpret by the parser. The description file contains four main sections:

- "*Devices*": where the digital devices are listed. Certain devices have the optional argument of the number of inputs to it which may also be specified
- "*Inputs*": where switches and a clock signal are initialised. All switches default to a `False` boolean value unless otherwise specified
- "*Connections*": The wiring from outputs to inputs of devices. Each device may appear in any order. Each device has its inputs initialised in any order.
- "*Monitor*": An optional section containing a list of the outputs of devices which should be displayed to the user. If this section is not included, the outputs of all devices will be shown to the user

Tabs-breaks are not considered part of the syntax of the definition file. The order of appearance of each section is important when parsing the language. Both "devices" and "inputs" must appear ahead of "connections". In addition, "devices" must appear before "monitor".

Curly brackets are used to delimit the different sections and nest subsections within a section. This will be useful in the implementation as they can be used 'checkpoints' during error handling.

Below is the grammar developed for our encryption language:

```
device_name =  letter, {letter | digit}

port_name = letter | digit, {letter | digit}

device_type = ‘‘CLOCK’’ | ‘‘SWITCH’’ | ‘‘DTYPE’’ | ‘‘NAND’’ | ‘‘NOR’’ | ‘‘XOR’’
```

```
| ‘‘AND’’ | ‘‘OR’’

connection = ‘‘to’’

definition = ‘‘is a’’ | ‘‘are’’ | ‘‘is’’ | ‘‘is an’’ | ‘‘are some’’

possession = ‘‘has’’ | ‘‘have’’

curly_open = ‘‘{‘‘

curly_closed = ‘‘}’’

dot =  ‘‘.’’

comma = ‘‘,’’

arrow = (‘‘=>’’ | ‘‘->’’)

colon = ‘‘:’’

range = device_name, arrow, device_name

device_definition = (device_name, {comma , device_name} | range),
definition, (device_type)

device_specification = (name, {comma , name} | range), possession,
number, [‘‘input’’ | ’’inputs’’]

switch_level = ‘‘set’’ , (device_name, {comma , device_name} | range)
, (‘‘HIGH’’ | ‘‘LOW’’)

clock_initialiser = (name, {comma , name} | range), ‘‘has cycle time
of’’, number

connect_definition = device_name, [‘‘BAR’’], connection, device_name,
dot, port_name

full_device_name = ‘‘Device’’, name
```

```
device_connection_definition = full_device_name , colon, curly_open,
connect_definition, {connect_definition}, curly_close

whole_devices = (‘‘devices’’ | ‘‘Devices’’), colon, curly_open, device_definition,
{device_definition}, device_specification, {device_specification},
curly_close

whole_init = (‘‘init’’ | ‘‘Init’’), colon, curly_open, device_definition,
{device_definition}, {clock_initailiser | switch_level}, {device_definition},
{clock_initailiser | switch_level}, curly_close

whole_connections = (‘‘connections’’ | ‘‘Connections’’), colon, curly_open,
device_connection_definition, {device_connection_definition}, curly_close

whole_monitor = (‘‘monitor’’ | ‘‘Monitor’’), colon, curly_open, (name
{comma, name} | range), {name {comma, name} | range}, curly_close

entirety = whole_devices, whole_init, whole_connections, whole_monitor
```

To illustrate the EBNF detailed, two examples have been provided with their circuit diagrams.

3.1 Example A - Simple circuit

The code that defines the circuit in figure 1 is given by:

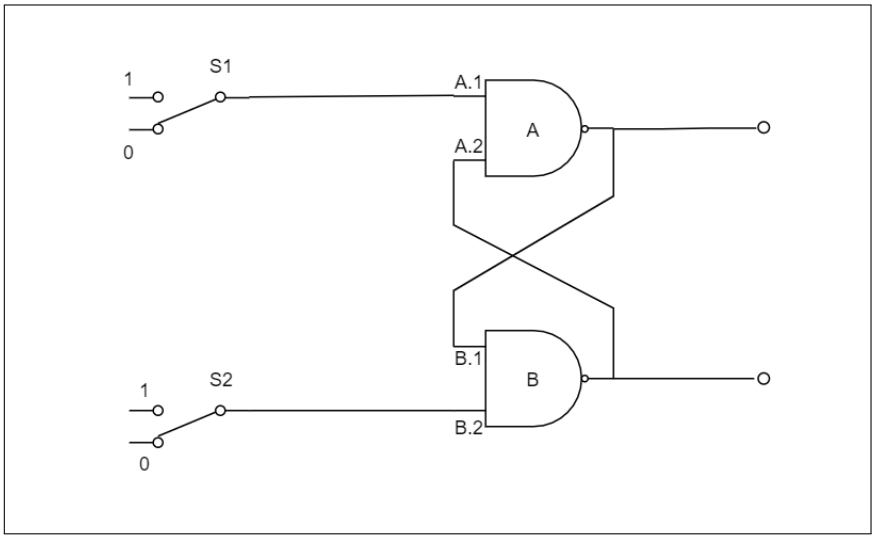

Figure 1: SR flip flip circuit

```
devices: {
    A,B are NAND gates
    A, B have 2 inputs
}
init: {
    S1, S2 are switches
    set S1, S2 low

connections: {
    device A: {
        S1 to A.1
        B to A.2
    }
    device B: {
        A to B.1
        S2 to B.2
    }
}
monitor: {
    A, B
}
```

3.2 Example B - Complex circuit

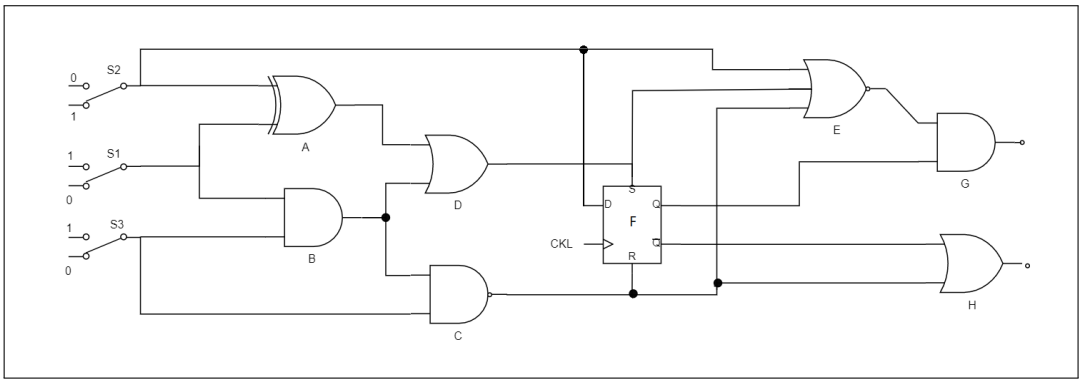

Figure 2: Trivial complex circuit

The logic circuit described in the figure 2 is trivial, and was constructed as an exercise to test the grammar rules of the language. Input gate names have been omitted for clarity, but it is noteworthy that which way round the inputs are do not matter other than for a D-type date, in which case the inputs are defined below, for a device named F:

Name	Gate
F.1	DATA
F.2	RESET
F.3	SET
F.4	CLOCK

The definition file for the circuit can be found below:

```
Devices: {
    A is an XOR gate
    B is a AND gate
    C is a NAND gate
    D is an OR gate
    E is NOR gate
    F is a DTYPE
    G is NAND
    H is OR
    A,B,C,D,G,H have 2 inputs
    E has 3 inputs
}

Init: {
    S1 => S3 are SWITCH
       inputs
    set S2 high
    CKL1 is a CLOCK input
    CKL1 has cycle time of 5
}

Connections: {
    Device A: {
        S2 to A.1
        S3 to A.2
    }
    Device B: {
        S1 to B.1
        S3 to B.2
    }
    Device C: {
        B to C.1
        S3 to C.2
    }
    Device D: {
        A to D.1
        B to D.2
    }
    Device E: {
        S2 to E.1
        D to E.2
        C to E.3
    }
    Device F: {
        S2 to F.1
        C to F.2
        D to F.3
        CLK1 to F.4
    }
    Device G: {
        E to G.1
        F to G.2
    }
    Device H: {
        not F to H.1
        C to H.2
    }
}

Monitor: {
    B, G, H
}
```

3.3 Error Identification and handling

Error identification is a key aspect of a user-friendly program. Therefore, there are several ways in which errors have been handled:

- Errors are raised immediately as the parser encounters them in the definition file.
- A useful message describing the nature and location (displaying the erroneous line, with a cursor indicating the location of the error) will be displayed to the user. This has been treated in a separate `error.py` module.
- Hints with what is expected in the error's location will be given

---