UNIVERSITY OF CAMBRIDGE
Department of Engineering

Part IIA Project: GF2(P2) Software

Ayoife Dada AOD24@CAM.AC.UK
Narmeephan Arunthavarajah NA596@CAM.AC.UK
Raghavendra Narayan Rao RN436@CAM.AC.UK
P2 Team 05
Homerton College
Department of Engineering
University of Cambridge
Cambridge
CB2 1PZ

Date: May 24, 2025

Abstract

Contents

1. Team Planning

1.1 Agile Scrum Framework

Our team adopted the Agile Scrum framework (2) to manage the project lifecycle. The process began with creating a project backlog, with all identified tasks required for project completion while remaining adaptable to new tasks as needed. Each weekly sprint commenced with a planning session, during which tasks were assigned to team members with clear deadlines outlined in the sprint plan. Daily Scrum meetings were conducted to monitor progress, address challenges, and ensure alignment. At the end of each sprint, incomplete tasks were reviewed and carried over to the subsequent sprint backlog, alongside any newly identified tasks. This iterative cycle will span three weeks/sprints for this project. Use the following link to follow out progress.

1.2 Git Branches

For this project, we implemented Git branching to efficiently manage code changes and maintain stable working versions. Branches allowed us to:

- Work on features without affecting the main branch
- Revert to previous versions if errors occurred
- Work simultaneously on separate branches before integration

By adopting this workflow, we ensured code stability while enabling continuous development and testing before finalizing changes in the main branch.

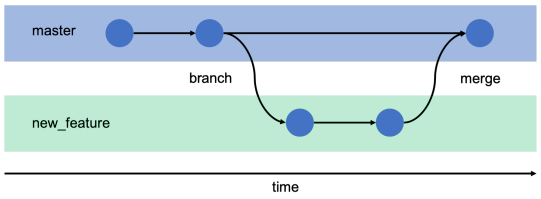

Figure 3: Visual representation of the git branch process

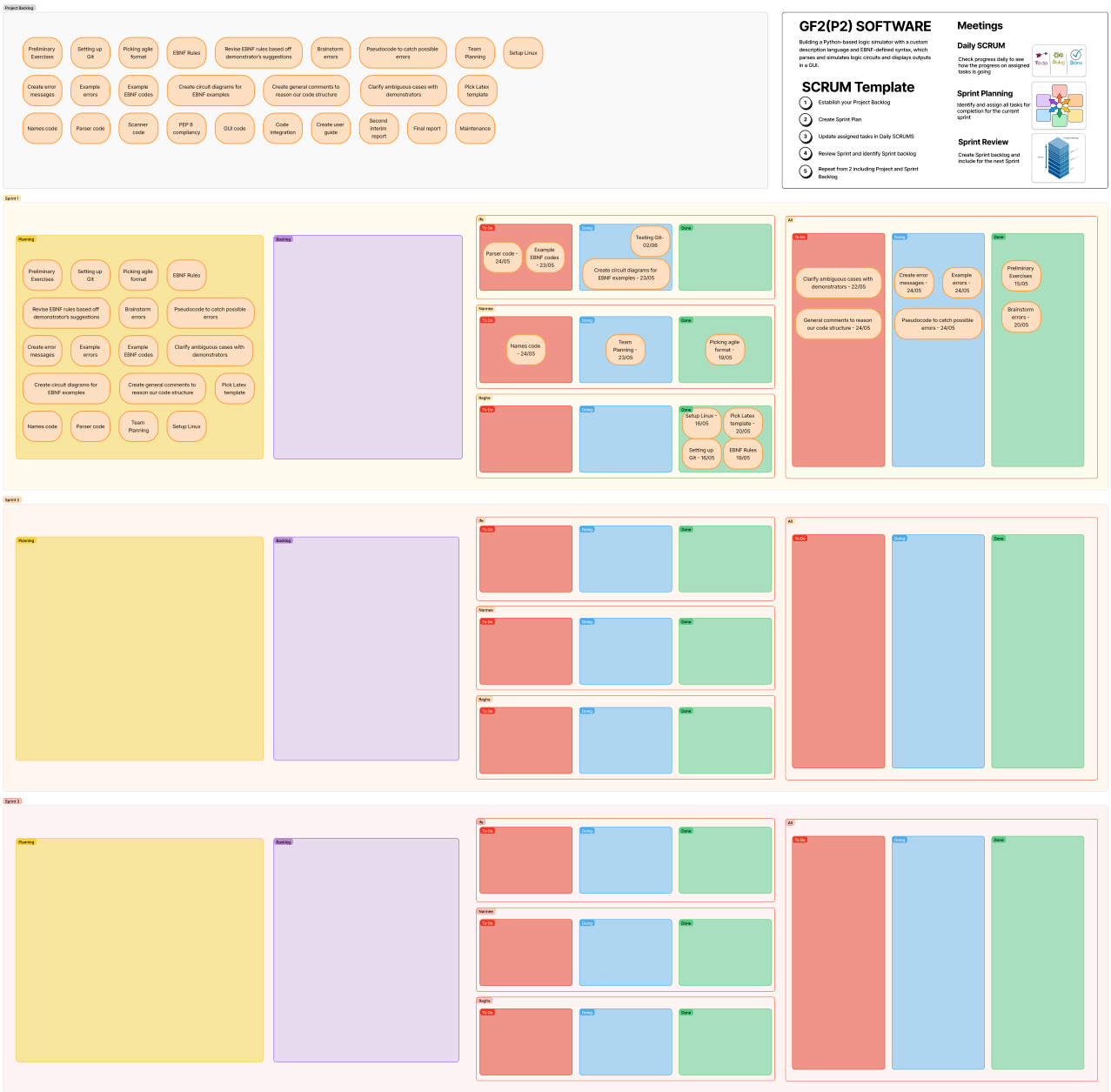

Figure 1: Project plan in the middle of sprint 1 (22/05)

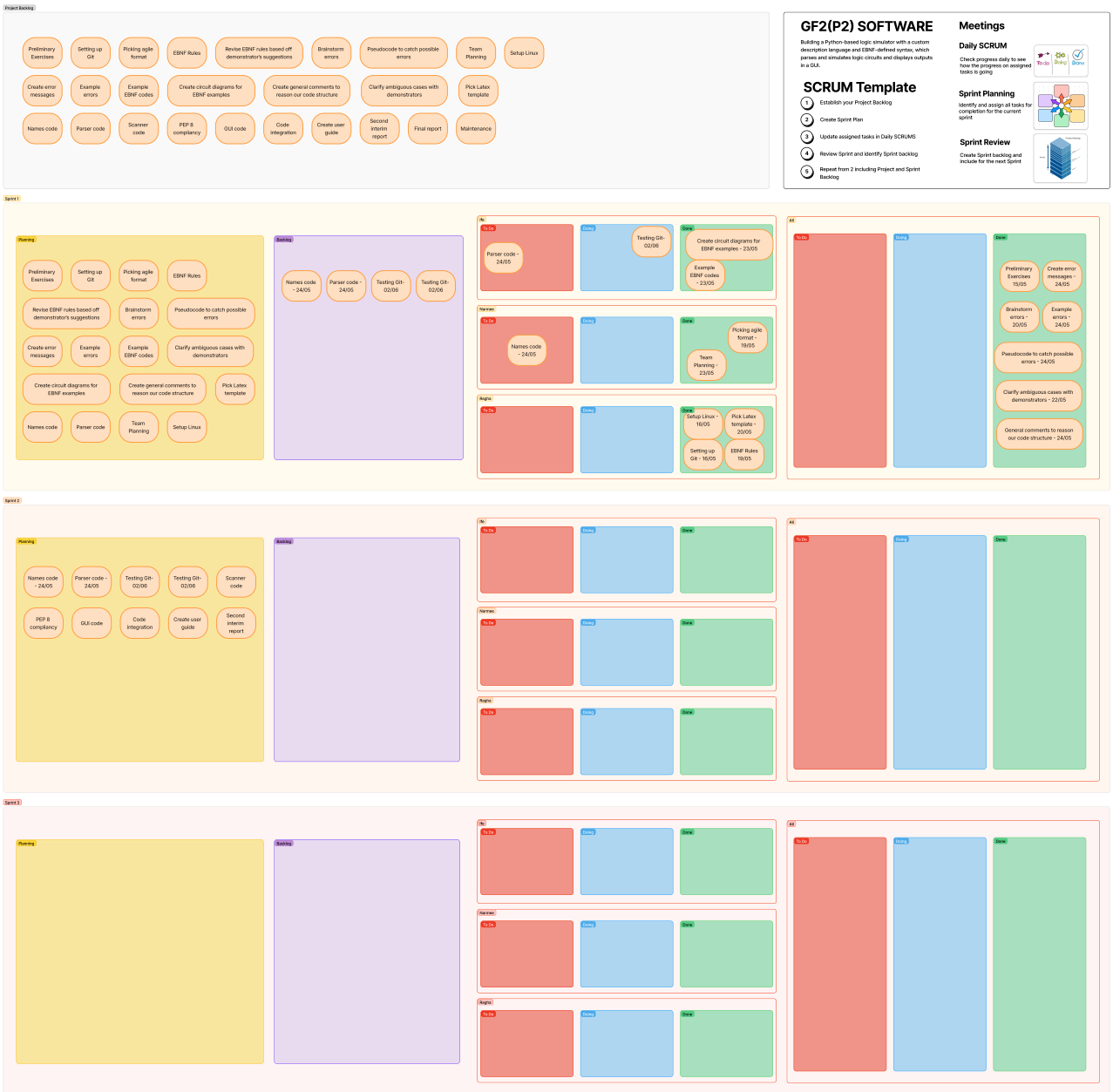

Figure 2: Project plan after sprint 1 (24/05)

2. EBNF Code

2.1 EBNF Rules

```
specfile = {comment}, devices, {devices | connection | monitor | comment} ;


devices =  "DEVICES ", device, {",", device} ,  eol ;

device = name,  ":",  devicetype, [ "(",  "IN",  ":",  number, ")"], [ "(",  "OUT",  ":",  number, ")"], [ "(",  "PERIOD",  ":",  number, ")"], [ "(",  "INITIAL",  ":",  bit, ")"]

devicetype = "CLOCK" | "SWITCH" | "AND" | "NAND" | "OR" | "NOR" | "DTYPE" | "XOR" ;


connection =  "CONNECT ", con, { ",",  con},  eol ;

con = signal,  "->",  signal ;

signal = name, ["." , pinname] ;

pinname = numberedpin | fixedpin ;

numberedpin = "I", ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" | "11" | "12" | "13" | "14" | "15" | "16") ;

fixedpin = "DATA"| "CLK" | "SET" | "CLEAR" | "Q" | "QBAR" ;


monitor =  "MONITOR ",  signal, { ",",  signal},  eol ;


comment = "/*", {" " | "\t" | "\n" | letter | digit | punctuation} "*/"


name = letter, {letter | digit} ;


eol = ";" ;

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

number = digit, {digit} ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

bit = "0" | "1" ;

punctuation = "." | "," | "!" | "?" | ":" | ";" | "'" | "(" | ")" | "[" | "]" | "{" | "}" | "-" | "&" | "@" | "#" | "%" | "^" | "*" | "+" | "=" | "<" | ">" | "|" | "~" | "$" ;
```

2.2 Design choices

1. Specfile
   Our EBNF rules ensure that devices are initialised before connections and monitors. Comments, however, can be placed anywhere. A good style is to handle different logic circuits one at a time instead of initialising all devices at once. Therefore, the "specfile" grammar rule provides flexibility of ordering code blocks. Here is an example of good style guide following our EBNF rules:

```
    /* A and B = C ------------------------ */

    /* initialise 2 SWITCHES and 1 AND gate */
    DEVICES A:SWITCH (INITIAL:0),
            B:SWITCH (INITIAL:0),
            C:AND (IN:2);

    /* Connect A and B to AND gate */
    CONNECT A -> C.I1,
            B -> C.I2;

    MONITOR C;


    /* D or E = F ------------------------ */

    /* initialise 2 SWITCHES and 1 OR gate */
    DEVICES D:SWITCH (INITIAL:0),
            E:SWITCH (INITIAL:0),
            F:OR (IN:2);

    /* Connect D and E to OR gate */
    CONNECT D -> F.I1,
            E -> F.I2;

    MONITOR F;
```

2. Device
   Contrary to common practice of defining syntax of each device seperately, our EBNF rule generalises to one common format. This increases the number of semantic errors to be handled (syntax errors converted to semantic errors) but allows for flexibility during maintenance. If new devices are added or device specifications are changed, we do not need to adapt our EBNF rules.

   Do note that this has allowed for the scenario where the programmer can specify unnecessary parameters but still be correct, such as a NAND gate having 1 output. This results in some redundancy in the code written.

```
    DEVICES N1:NAND (IN:11) (OUT:1);
```

3. Pinname
   By manually defining the numbered pins from 1 to 16 as opposed to any number, potential semantic errors have been converted to syntax errors. Syntax errors are on average caught quicker than

semantic errors. There are two reasons for this. Firstly, syntax errors are on average caught in $O(1)$ time while semantic errors can take $O(N)$ due to further loops required. Secondly, syntax errors are usually caught exactly when they are encountered but semantic errors might be caught later during compilation.

4. Monitor
   Our EBNF rules allow for a signal to be monitored multiple times. This design choice provides better visualisation in situations where a signal has to be compared with multiple other signals. To demonstrate this, here is an example comparing two situations, with and without monitor restrictions on repeated signals.

   (Situation 1) Disallow repeated signals to be monitored:

```
MONITOR N1, A1, B1, C1;
```

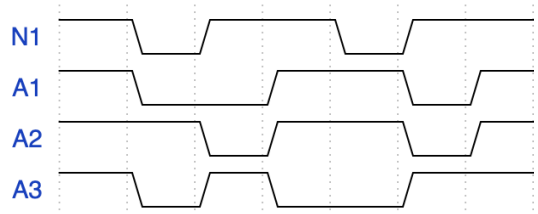

Figure 4: (Situation 1) Example plots of signals in GUI

(Situation 2) Allow repeated signals to be monitored:

```
MONITOR N1, A1;
MONITOR N1, B1;
MONITOR N1, C1;
```

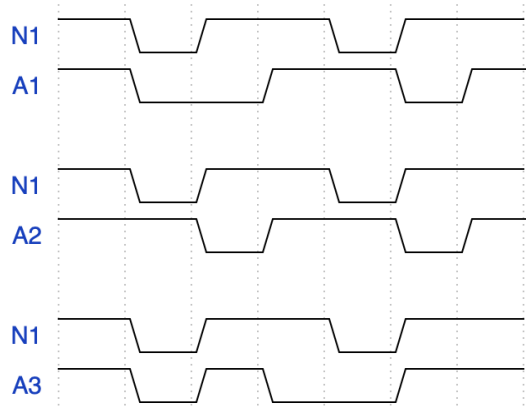

Figure 5: (Situation 2) Example plots of signals in GUI

Notice how it is much easier to visualise the difference between signals N1 and A3 in situation 2. Therefore, monitoring repeated signals has been allowed.

2.3 EBNF Example Code with diagrams

1. EBNF for simple logic circuit using D-type flip flops and a NAND gate.

```
    /*
    Name the gates, switches and clocks
    Number of inputs and outputs does not need to be defined for DTYPE
    */

    DEVICES D1:DTYPE,
            D2:DTYPE,
            N1:NAND (IN:2),
            C1:CLOCK (PERIOD:8),
            S1:SWITCH (INITIAL:0),
            S2:SWITCH (INITIAL:1),
            S3:SWITCH (INITIAL:0) ;

    /* connect inputs and outputs */
    CONNECT S1->D1.SET,
            S1->D2.SET,
            S2->D1.DATA,
            S3->D1.CLEAR,
            S3->D2.CLEAR,

            C1->D1.CLK,
            C1->D2.CLK,

            D1.Q->D2.DATA,
            D2.Q->N1.I1,
            D2.QBAR->N1.I2;

    /* monitor certain signals */
    MONITOR D1.QBAR,
            N1;
```

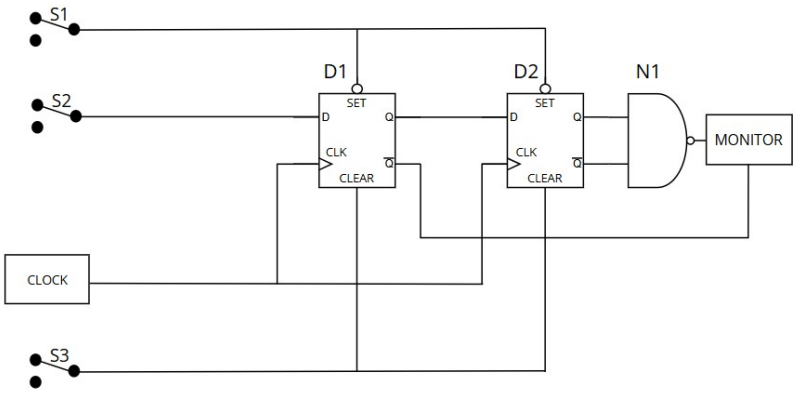

Figure 6: Visual logic circuit design for EBNF (example one) (1)

2. EBNF using XOR, AND and OR gates to create fuller-adder circuit.

```
    /* This configuration is a full-adder */
    ---------------------------------------

    /* name all devices */
    DEVICES X1:XOR (IN:2),
            X2:XOR (IN:2),
            A1:AND (IN:2),
            A2:AND (IN:2),
            O1:OR (IN:2),
            S1:SWITCH (INITIAL:1),
            S2:SWITCH (INITIAL:1),
            S3 SWITCH (INITIAL:0);

    /* connect inputs and outputs */
    CONNECT S1->X1.I1,
            S1->A1.I1,
            S2->X1.I2,
            S2->A1.I2,
            S3->X2.I2,
            S3->A2.I2,

            X1->X2.I1,
            X1->A2.I1,

            A1->O1.I1,
            A2->O1.I2;

    /* monitor particular signals */
    MONITOR X2,
            O1;
```

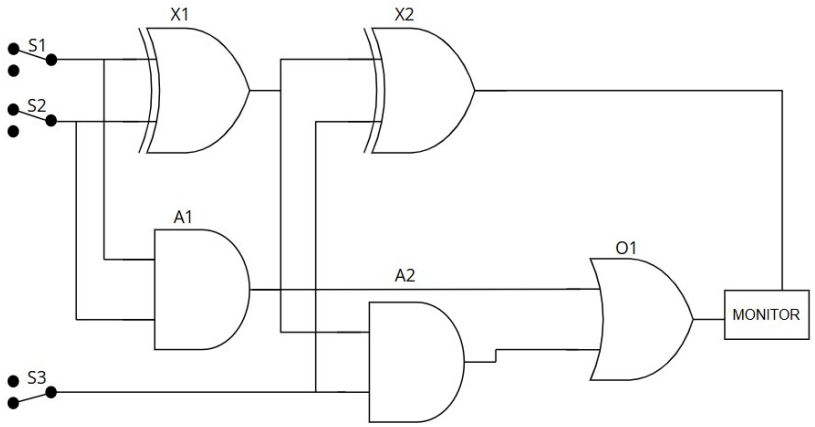

Figure 7: Visual logic circuit design for EBNF (example two) (1)

2.4 EBNF Cursed Example Code

The following code shows that our language is free-form so spaces and new lines do not affect the code validity. Hence, the code is identical to the second EBNF example code.

```
    /* This code is an extreme example to showcase the
    free-form nature of the language. */
    ------------------------------------------------

    /* name all                  devices */
    DEVICES X1:XOR (IN:2),
                          X2:XOR (IN:2),
            A1:AND (IN:2),
A2:AND (IN:2),
O1:OR (IN:2),
            S1:SWITCH (INITIAL:1),
        S2:SWITCH (INITIAL:1), S3 SWITCH (INITIAL:0) ;

    /* connect inputs


    and outputs */
    CONNECT S1->                X1.I1,
            S1 ->A1.I1,
        S2->X1.I2,
            S2 -> A1.I2,
S3->X2.I2   ,
S3->A2.I2,

            X1->X2.I1,
            X1->  A2.I1,A1->O1.I1,A2->O1.I2 ;

    /* monitor particular signals */
    MONITOR



    X2,
            O1 ;
```

3. Error Handling

3.1 End of Line character

An "end of line" (eol) character, specifically a semicolon, has been added at the end of every executable code. When the compiler (parser and scanner) encounters a syntax error, it skips forward to the semicolon (or "*/" for multiline-comments) and continues compiling the rest of the code. Here in an example that demonstrates how the compiler parses the code. Text in green represents code successfully parsed by the compiler while text in black represents code that has been skipped by the compiler.

```
1   /* A and B = C/-------------------- */
2   /* initialise 2 SWITCHES and 1 AND gate */
3   DEVICES A:SWITCH (INITIAL:0),
4            B:SWICH (INITIAL:0),
5            C:AND (IN:2);
6
7   CONNECT A --> C.I1,
8           B -> C.I2;
9
10  MONITOR F;
```

1. Line 1: '/' is not allowed in comments.
2. Line 4: SWITCH has been spelt incorrectly as SWICH.
3. Line 7: CONNECT uses '− >' for connecting signals but '−− >' has been used instead.
4. Line 10: F is an undeclared device.

3.2 Useful Error Message

After the compiler has scanned through the entire file, an additional feature will be added to the parser that classifies all errors into general issues related to devices, connections or monitors. Within each category, specific errors will be listed in the order that they are encountered, along with suggested fixes. This will allow programmers to view and fix similar errors efficiently. Here is an example template of an error message.

```
There are 16 errors.

Errors pertaining to devices: 7

    1. Invalid Parameter Order: Expected IN before OUT (line 5).

       DEVICES A1:AND (OUT:1) (IN:2);

    2. Invalid Device Type: There is no such device as AAND. (line 7)

       DEVICES A1:AAND (IN:5);

    ...

Errors pertaining to connections: 5

    1. Invalid connections: S1, a SWITCH, has no input (line 21).

       CONNECT A1 -> S1.I1,

    2. Invalid connections: A1 is connected to itself (line 25).

       CONNECT A1 -> A1.I1,

    ...

Errors pertaining to monitors: 4

    1. Invalid devicename: F1 is undeclared (line 30).

       MONITOR F1;

    2. Invalid devicename: D1 only has CLK, DATA, SET, CLEAR, Q and QBAR pins (line 32).

       MONITOR D1.I1;

    ...
```

4. Syntax Errors

Syntax errors occur when the written code explicitly violates the EBNF rules. While a list of syntax errors is not required in this report, certain interesting syntax errors that can occur under our EBNF rules is provided below.

4.1 Missing Semicolons

```
1    DEVICES N1:NAND (IN:11)
2    CONNECT S1 -> A1.I1,
3            C1 -> A1.I7;
```

This syntax error occurs when a semicolon is missing at the end of an executable line of code. Here is the error message that will be displayed:

```
Missing semicolon: DEVICES N1:NAND (IN:11);
```

However, catching this syntax error is not trivial. A naive answer might be to raise an error if an initialisation keyword ("DEVICES", "CONNECT", "MONITOR", "") is encountered before a semicolon. The following pseudocode illustrates how this error will be caught:

Algorithm 1 Check for valid end-of-line semicolon (naive)

Let *initialisation_keywords* ← { "DEVICES", "CONNECT", "MONITOR" }
while *isSemicolon* is **False do**
  if *parsed_word* ∈ *initialisation_keywords* **then**
    raise error "Missing semicolon: {*current_line*};"
  end if
  update *isSemicolon*
end while

This algorithm fails to identify the missing semicolon error when the programmer spells the keywords incorrectly. The compiler will misinterpret that the device has been initialized incorrectly and will raise a syntax error, as it is expecting the next parameter.

```
1    DEVICES N1:NAND (IN:11)
2    CONECT S1 -> A1.I1,
3           C1 -> A1.I7;
```

```
Invalid Device Initialisation: Expected a "("
```

In order to accurately identify a missing semicolon error, the compiler will need to understand when an executable line of code has been completely specified and hence expect a semicolon. This is simply not possible with hard coding. Some level of intelligence will be required.

To demonstrate the difficulty of this problem, here is an example from one of the oldest modern languages, C, that still misidentifies a missing semicolon error. The following code is a classic swapping algorithm that swaps 2 numbers by using a temporary variable.

```c
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = 0;
    temp = *a  // Oops, missing a semicolon here
    *a = *b;
    *b = temp;
}

int main(void) {
    int a = 1;
    int b = 2;
    printf("Before Swap) a: %d, b: %d\n", a, b);
    swap(&a, &b);
    printf("After Swap)  a: %d, b: %d\n", a, b);
}
```

With the semicolon, the swap occurs successfully with the following standard output:

```
Before Swap) a: 1, b: 2
After Swap)  a: 2, b: 1
```

Without the semicolon, the following misidentified error is raised.

```
invalid operands to binary * (have 'int' and 'int *')
```

Since C is a free-form language, the following expressions are equivalent.

```c
temp = *a
*a = *b;

temp = *a * a = *b;  // interpreted as (*a) multipled with a
```

4.2 Invalid Parameter Order

```
DEVICES A1:AND (OUT:1) (IN:2);
```

```
Invalid Parameter Order: Expected IN before OUT
```

This syntax error occurs because we have implemented a specific order to parameters in our EBNF rules. The "OUT" parameter is redundant because AND gates only have one output, but we have allowed for the programmer to specify it anyway as long as it is correct (read more about our design choices in Section 2).

The interesting aspect of this syntax error is how it can be prevented using EBNF specification. Ideally, a programmer should be allowed to specify the parameters of a general device in any order. However, this will require specifying $2^4 = 16$ different possible EBNF rules. This is not scalable with number of input parameters.

```
device = name, ":", devicetype, [ "(", "IN", ":", number, ")"], [ "(", "OUT",
    ":", number, ")"], [ "(", "PERIOD", ":", number, ")"], [ "(", "INITIAL", ":
    ", bit, ")"]

device = name, ":", devicetype, [ "(", "OUT", ":", number, ")"], [ "(", "IN",
    ":", number, ")"], [ "(", "PERIOD", ":", number, ")"], [ "(", "INITIAL", ":
    ", bit, ")"]

...
```

Another option is to convert specifications to be optional. However, this EBNF allows for repeated parameter definition that can contradict each other. For example, the number of input parameters can be defined multiple times, possibly different, increasing the number of semantic errors. Therefore, the EBNF rule below was not chosen.

```
device = name, ":", devicetype, parameter, parameter, parameter, parameter, eol ;

parameter = [ "(", "IN", ":", number, ")"] | [ "(", "OUT", ":", number, ")"] |
    [ "(", "PERIOD", ":", number, ")"] | [ "(", "INITIAL", ":", bit, ")"] ;
```

5. Semantic Errors

5.1 Invalid pin number

```
    DEVICES A1:AND (IN:2),
            S1:SWITCH (INITIAL:0),
            C1:CLOCK (PERIOD:5);
    CONNECT S1 -> A1.I1,
            C1 -> A1.I7;
```

This semantic error occurs when the input pin number of a device exceeds the number of inputs specified for the device. Since our EBNF code allows for any pin number that is between 1 and 16, this error will not be caught as a syntax error. Here is the error message that will be displayed:

```
Invalid pin number: inputs must be in the range of 1 to <number of defined inputs>
```

The following pseudocode illustrates how this error will be caught:

Algorithm 2 Check Pin Number Validity

input_num ← getInputNumber(*device.pinname*)
if *input_num* > *device.num_inputs* **then**
    raise "Invalid input number: inputs must be in the range of 1 to *device.num_inputs*"
end if

5.2 Invalid input/output number

```
DEVICES C1:CLOCK (OUT:2);
```

```
DEVICES C1:CLOCK (IN:1);
```

```
DEVICES S1:SWITCH (OUT:2);
```

```
DEVICES S1:SWITCH (IN:1);
```

```
DEVICES A1:AND (OUT:2);
```

```
DEVICES D1:DTYPE (OUT:1);
```

```
DEVICES D1:DTYPE (IN:5);
```

```
DEVICES X1:XOR (IN:3);
```

This semantic error occurs when the input or output number for a device is incorrect. The allowed input and outputs for each devices are listed in the handout as follows:

Component	Outputs	Inputs
CLOCK	1	0
SWITCH	1	0
AND NAND OR NOR	1	1-16
DTYPE	2	4
XOR	1	2

Table 1: Logic Component Specifications

Since our EBNF code allows for providing these parameters regardless of the device type, this is not a syntax error.

5.2.1 Outputs

Here are the error messages that will be displayed:

```
Invalid output number: number of outputs for a CLOCK must always be 1
Invalid output number: number of outputs for a SWITCH must always be 1
Invalid output number: number of outputs for a AND, NAND, OR, NOR, XOR must always be 1
Invalid output number: number of outputs for a DTYPE must always be 2
```

Algorithm 3 Check Output Number Validity

$CLOCK\_output\_num \leftarrow$ getOutputNumber($C1.pinname$)
if $CLOCK\_output\_num \neq 1$ **then**
    raise "Invalid output number: number of outputs for a CLOCK must always be 1"
end if

Algorithm 4 Check Output Number Validity

$SWITCH\_output\_num \leftarrow$ getOutputNumber($S1.pinname$)
if $SWITCH\_output\_num \neq 1$ **then**
    raise "Invalid output number: number of outputs for a SWITCH must always be 1"
end if

Algorithm 5 Check Output Number Validity

$GATE\_output\_num \leftarrow$ getOutputNumber($G1.pinname$)
if $GATE\_output\_num \neq 2$ **then**
    raise "Invalid output number: number of outputs for a gate must always be 1"
end if

Algorithm 6 Check Output Number Validity

$DTYPE\_output\_num \leftarrow$ getOutputNumber($D1.pinname$)
if $DTYPE\_output\_num \neq 2$ **then**
    raise "Invalid output number: number of outputs for a DTYPE must always be 2"
end if

5.2.2 Inputs

Here are the error messages that will be displayed:

```
Invalid input number: number of inputs for a CLOCK must always be 0
Invalid input number: number of inputs for a SWITCH must always be 0
Invalid input number: number of inputs for a DTYPE must always be 4
Invalid input number: number of inputs for a XOR must always be 2
```

Algorithm 7 Check Input Number Validity

$CLOCK\_input\_num \leftarrow$ getInputNumber($C1.pinname$)
if $CLOCK\_input\_num \neq 0$ **then**
    raise "Invalid input number: number of inputs for a CLOCK must always be 0"
end if

Algorithm 8 Check Input Number Validity

$SWITCH\_input\_num \leftarrow$ getInputNumber($S1.pinname$)
if $SWITCH\_input\_num \neq 0$ **then**
    raise "Invalid input number: number of inputs for a SWITCH must always be 0"
end if

Algorithm 9 Check Input Number Validity

$DTYPE\_input\_num \leftarrow$ getInputNumber($D1.pinname$)
if $DTYPE\_input\_num \neq 4$ **then**
    raise "Invalid input number: number of inputs for a DTYPE must always be 4"
end if

Algorithm 10 Check Input Number Validity

$XOR\_input\_num \leftarrow$ getInputNumber($X1.pinname$)
if $XOR\_input\_num \neq 2$ **then**
    raise "Invalid input number: number of inputs for a XOR must always be 2"
end if

5.3 Invalid device name

```
1     DEVICES CLOCK:XOR;
```

This semantic error occurs when a device is given the name of a keyword. The keyword are as follows: DEVICES, CONNECT, MONITOR, IN, OUT, PERIOD, INITIAL, CLOCK, SWITCH, AND, NAND, OR, NOR, DTYPE, XOR, DATA, CLK, SET, CLEAR, Q and QBAR. Our EBNF rules allow any name that starts with a letter and then a combination of numbers and letters, therefore this is not a syntax error.

Here is the error message that will be displayed:

```
Invalid device name: devices can not have a name which is a keyword
```

The following pseudocode illustrates how this error will be caught:

Algorithm 11 Check Device Name Validity

Let *keywords* ← { “DEVICES”, ... ,“QBAR” }
if *device[i].name* ∈ *keywords* **then**
    raise error “Invalid device name: devices can not have a name which is a keyword”
end if

5.4 Invalid Parameters

```
1    DEVICES A1:AND (PERIOD:2);
```

This semantic error occurs when a parameter that the device does not contain is given. The parameters that each device contain is as follows:

Component	IN	OUT	PERIOD	INITIAL
CLOCK	☒	☒	☒	☐
SWITCH	☒	☒	☐	☒
AND/NAND/OR/NOR	☒	☒	☐	☐
DTYPE	☒	☒	☐	☐
XOR	☒	☒	☐	☐

Table 2: Component Parameters

Since defining each parameter is allowed regardless of the device type, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid parameters: <device> does not contain the parameter <invalid parameter>
```

The following pseudocode illustrates how this error will be caught:

Algorithm 12 Check Parameter Validity

Let *allowed_parameters* ← {"CLOCK": {"IN", "OUT", "PERIOD"},
... : {...},
"XOR": {"IN", "OUT"}}
if *device.parameter* ∉ *allowed_parameters* [*device.type*] **then**
raise error "Invalid parameters: *device* does not contain the parameter *device.parameter*"
end if

```
1    DEVICES C1:CLOCK (IN:0);
```

This semantic error occurs when required parameters are not defined. Required parameters are as follows:

Table 3: Required Parameters Checklist

Component	IN	OUT	PERIOD	INITIAL
CLOCK	☐	☐	☒	☐
SWITCH	☐	☐	☐	☒
AND/NAND/OR/NOR	☒	☐	☐	☐
DTYPE	☐	☐	☐	☐
XOR	☐	☐	☐	☐

Since it is not required to provide any specific parameter values, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid parameters: <required parameter> needs to be defined for <device>
```

The following pseudocode illustrates how this error will be caught:

Algorithm 13 Check Parameter Validity

Let *required_parameters* ← {"CLOCK": {"PERIOD"},
... : {...},
"XOR": {"IN"}}
if *required_parameters* [*device.type*] ∉ *device.parameter* **then**
raise error "Invalid parameters: *required_parameters* [*device.type*] needs to be defined for *device*"
end if

5.5 Invalid Connections

```
    DEVICES C1:CLOCK (PERIOD:4),
            S1:SWITCH (INITIAL:1);

    CONNECT C1 -> S1;
```

This semantic error occurs when a connection is made to a device with no input, CLOCKS and SWITCHES. Since any device can be connected to any other device according to the language rules, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid connections: <device> has no input
```

The following pseudocode illustrates how this error will be caught:

Algorithm 14 Check Connection Validity

```
output_device ← getDeviceName(connection[i].out)
no_input_devices ← {"CLOCK", ..., "SWITCH"}
if output_device.type ∈ no_input_devices then
    raise error "Invalid connections: output_device has no input"
end if
```

```
    DEVICES C1:CLOCK (PERIOD:4),
            A1:AND (IN:2);

    CONNECT C1 -> A1.I1,
            A1 -> A1.I2;
```

This semantic error occurs when a connection is made to itself. Since any device can be connected to any other device, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid connections: <device> is connected to itself
```

The following pseudocode illustrates how this error will be caught:

Algorithm 15 Check Connection Validity

Let *output_device* ← getDeviceName(*connection*[*i*].*out*)
if *connection*[*i*].*in* = *connection*[*i*].*out* **then**
    raise error "Invalid connections: output_device is connected to itself"
end if

```
    DEVICES C1:CLOCK (PERIOD:4),
            S1:SWITCH (INITIAL:1),
            A1:AND (IN:2);

    CONNECT C1 -> A1.I1,
            S1 -> A1.I1;
```

This semantic error occurs when multiple signals are connected to the same input. Since any device can be connected to any other device, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid connections: Multiple connections to <input location>
```

The following pseudocode illustrates how this error will be caught:

Algorithm 16 Check Connection Validity

Let *connection[i]* $\leftarrow$ ith connection
if $\exists\ i \neq j$ such that $connection[i].out = connection[j].out) \wedge (connection[i].in \neq connection[j].in)$ **then**
    raise error "Invalid connections: Multiple connections to connection[i]"
end if

5.6 Invalid Pinname

5.6.1 Clocks and switches

```
    DEVICES C1:CLOCK (PERIOD:2),
            S1:SWITCH (INITIAL:1),
            A1:AND (IN:2);

    CONNECT C1.I1 -> A1.I1,
            S1 -> A1.I2;
```

This semantic error occurs when a pinname is provided for a device with no inputs. Since pinname can be provided regardless of the device, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid pinname: <device> has no pins
```

The following pseudocode illustrates how this error will be caught:

Algorithm 17 Check Pinname Validity

Let *connection[i]* ← ith connection
Let *input_device* ← getDeviceName(*connection[i].in*)
Let *devices_with_no_input* ← {"CLOCK", ..., "SWITCH"}
if (*input_device.type* ∈ *devices_with_no_input*) ∧ hasPin(*input_device*) **then**
    raise error "Invalid connections: input_device has no pins"
end if

5.6.2 D-types and gates

```
    DEVICES C1:CLOCK (PERIOD:2),
            S1:SWITCH (INITIAL:1),
            S2:SWITCH (INITIAL:0),
            S3:SWITCH (INITIAL:0),
            D1:DTYPE;

    CONNECT C1 -> D1.CLK,
            S1 -> D1.DATA,
            S2 -> D1.SET,
            S3 -> D1.I1;
```

```
    DEVICES C1:CLOCK (PERIOD:2),
            S1:SWITCH (INITIAL:1),
            A1:AND (IN:2);

    CONNECT C1 -> A1.I1,
            S1 -> A1.CLK;
```

This semantic error occurs when a pinname is provided for a device without that pin. Gates have input pins of the form I followed by a number. DTYPEs have input pins of CLK, DATA, SET and CLEAR.

Therefore, setting CLK pins for gates and I{digit} for DTYPEs is an error. Since pinname can be provided regardless of the device, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid pinname: <DTYPE_name> only has CLK, DATA, SET and CLEAR pins
Invalid pinname: <GATE_name> only has input pins
```

The following pseudocode illustrates how this error will be caught:

Algorithm 18 Check Pinname Validity

Let *connection[i]* ← ith connection

Let $DTYPE\_input\_pins \leftarrow \{DATA, CLK, SET, CLEAR\}$
Let *input_device* ← getDeviceName(*connection*[*i*].*in*)
Let *input_pin* ← getInputPinName(*connection*[*i*].*in*)
if $(input\_device.type = DTYPE) \wedge (input\_pin \notin DTYPE\_input\_pins)$ **then**
  raise error "Invalid pinname: input_device only has CLK, DATA, SET and CLEAR pins"
end if

Let $DTYPE\_output\_pins \leftarrow \{Q, QBAR\}$
Let *output_device* ← getDeviceName(*connection*[*i*].*out*)
Let *output_pin* ← getOutputPinName(*connection*[*i*].*out*)
if $(output\_device.type = DTYPE) \wedge (output\_pin \notin DTYPE\_ouput\_pins)$ **then**
  raise error "Invalid pinname: output_device only has Q and QBAR pins"
end if

Algorithm 19 Check Pinname Validity

Let *connection[i]* ← ith connection

Let $gate\_list \leftarrow \{AND, NAND, OR, NOR, XOR\}$
Let *input_device* ← getDeviceName(*connection*[*i*].*in*)
Let *input_pin* ← getInputPinName(*connection*[*i*].*in*)
if $(input\_device.type \in gate\_list) \wedge (input\_pin[0] \neq "I")$ **then**
  raise error "Invalid pinname: input_device only has input pins"
end if

Let *output_device* ← getDeviceName(*connection*[*i*].*out*)
Let *output_pin* ← getOutputPinName(*connection*[*i*].*out*)
if $(output\_device.type \in gate\_list) \wedge (output\_pin[0] \neq "I")$ **then**
  raise error "Invalid pinname: output_device only has input pins"
end if

5.7 Undeclared Device

```
DEVICES S1:SWITCH (INITIAL:1),
        A1:AND (IN:2);

CONNECT C1 -> A1.I1,
        S1 -> A1.I2;

MONITOR A2;
```

This semantic error occurs when a devicename is used it has not been declared. Since any devicename that starts with a letter is valid, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid devicename: <devicename> is undeclared
```

The following pseudocode illustrates how this error will be caught:

Algorithm 20 Check whether Devicename Undeclared in Connections

Let *connection[i]* ← ith connection

Let *input_device* ← getDeviceName(*connection*[*i*].*in*)
if (*devicename* ≠ *list_of_devicenames*) **then**
    raise error "devicename is undeclared."
end if

Let *output_device* ← getDeviceName(*connection*[*i*].*out*)
if (*devicename* ≠ *list_of_devicenames*) **then**
    raise error "devicename is undeclared."
end if

Algorithm 21 Check whether Devicename Undeclared in Monitor

Let *monitor[i]* ← ith monitored signal
Let *input_device* ← getDeviceName(*monitor*[*i*])
if (*devicename* ≠ *list_of_devicenames*) **then**
    raise error "devicename is undeclared."
end if

5.8 Invalid Monitor Connection

```
DEVICES S1:SWITCH (INITIAL:1),
        S2:SWITCH (INITIAL:0),
        C1:CLOCK (Period:5),
        D1:DTYPE;

CONNECT S1 -> D1.DATA,
        S1 -> D1.SET,
        S2 -> D1.CLEAR,
        C1 -> D1.CLK;

MONITOR D1;
```

This semantic error occurs when the monitored signal cannot be uniquely identified. This only occurs when a device has multiple outputs i.e. d-type flip flop output to be monitored is not specified. Since the monitor can be connected to any signal, this is not a syntax error. Here is the error message that will be displayed:

```
Invalid monitored pin connection: <device> has no <pin>.
Device pins should be one of  DATA, CLK, SET, CLEAR, Q or QBAR''
```

The following pseudocode illustrates how this error will be caught:

Algorithm 22 Check Pinname Validity

Let *monitor[i]* ← ith monitored signal
Let $DTYPE\_pins \leftarrow \{DATA, CLK, SET, CLEAR, Q, QBAR\}$
Let *device* ← getDeviceName(*monitor*[*i*])
Let *pin* ← getInputPinName(*monitor*[*i*])
if $(device.type = DTYPE) \wedge (pin \notin DTYPE\_pins)$ **then**
    raise error "Invalid monitored pin: *device* has no *pin*, ,device pins should be one of DATA, CLK, SET, CLEAR, Q or QBAR"
end if

5.9 Too Few Connections

```
DEVICES S1:SWITCH (INITIAL:1),
        A1:AND (IN:2);

CONNECT S1 -> A1.I1;

MONITOR A1;
```

This semantic error occurs when the number of connections to a device is less than the number of inputs defined in the device parameters. Since the entire code must be compiled before an error can be detected, this is a semantic error. Here is the error message that will be displayed:

```
Too few connections to <devicename>: number of inputs connected should be equal to
number of inputs defined for device.
```

The following pseudocode illustrates how this error will be caught:

Algorithm 23 Check Number of Input Connections

Let *output_signals* ← set of unique output signals from CONNECT
Let *number_of_inputs* ← 0
for *device* ∈ list of devices **do**
    number_of_inputs ← getUniqueInputConnections(*output_signals*, *device.name*)
    if *number_of_inputs* ≠ *device.num_inputs* **then**
        raise error "Not enough connections to device.name."
    end if
end for

5.10 Repeated Device Names

```
1     DEVICES S1:SWITCH (INITIAL:1);
2     DEVICES S1:AND (IN:2);
```

This semantic error occurs when another device is declared with the same name. In python, this would change the variable but in our language this raises an error because changing a device's parameters would require the compiler to re-check all connections. Since no naming conventions are violated, this is not a syntax error. Here is the error message that will be displayed:

```
Repeated Device Names: <devicename> is repeated, do not name multiple devices with the same name
```

The following pseudocode illustrates how this error will be caught:

Algorithm 24 Check whether Device Name repeated

Let *device_names* ← current set of unique device names
if *current_device_name* ∉ *device_names* **then**
    raise error "Repeated Device Name: current_device_name is repeated, do not name multiple devices with the same name"
end if

6. General Comments

6.1 LL(1)

LL(1) framework refers to only requiring lookup of 1 token to make a choice between multiple available options. Below is a comprehensive list of all rules that require a choice. The following categorisation will prove that our code follows LL(1).

1. Direct Lookup.

```
devicetype = "CLOCK" | "SWITCH" | "AND" | "NAND" | "OR" | "NOR" | "DTYPE" | "XOR" ;

numberedpin = "I", ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" | "11
    " | "12" | "13" | "14" | "15" | "16") ;

comment = "/*", {" " | "\t" | "\n" | letter | digit | punctuation} "*/"

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
     | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
     "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "
    n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

bit = "0" | "1" ;

punctuation = "." | "," | "!" | "?" | ":" | ";" | "'" | "(" | ")" | "[" | "]" | "{"
    | "}" | "-" | "&" | "@" | "#" | "%" | "^" | "*" | "+" | "=" | "<" | ">" | "|" |
    "~" | "$" ;
```

2. Lookup of 1 token: specfile

```
specfile = {comment}, devices, {devices | connection | monitor | comment};

devices =  "DEVICES ", device, {",", device} ,  eol ;
connection =  "CONNECT ", con, { ",",  con},  eol ;
monitor =  "MONITOR ",  signal, { ",",  signal},  eol;
comment = "/*", {" " | "\t" | "\n" | letter | digit | punctuation} "*/"
```

These 4 blocks can be distinguished with the first keyword (DEVICES, CONNECT, MONITOR, */)

3. Lookup of 1 token: pinname

```
pinname = numberedpin | fixedpin ;

numberedpin = "I", ("1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "10" | "11
    " | "12" | "13" | "14" | "15" | "16") ;
fixedpin = "DATA"| "CLK" | "SET" | "CLEAR" | "Q" | "QBAR" ;
```

There is no overlap between numbered pins and fixed pins (numbers and alphabets) so this is LL(1).

4. Lookup of 1 token: name

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

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M"
     | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
     "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "
    n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

There is no overlap between digits and letters so this is LL(1).

6.2 Output signal

```
    DEVICES S1:SWITCH (INITIAL:1),
            C1:CLOCK (Period:5),
            D1:DTYPE,
            N1: NAND (IN:5);
    ...
    MONITOR D1.QBAR, S1, C1, N1;
```

Notice how the output signals being monitored are labeled as the device name itself, except for D1, which is a DTYPE with 2 outputs. This is a convention we have adopted to simplify readability of code. Of course, a seperate output signal could have been defined under EBNF rule to indicate an OUT signal, resulting in the following convention:

```
    MONITOR D1.QBAR, S1.OUT, C1.OUT, N1.OUT;
```

While being equally valid, this introduces redundancy in the programmer's code. Therefore, we have adopted the first convention listed.

6.3 No Initial Value for CLOCK

We have chosen to raise an error when an initial value is defined for a clock. Here are two reasons:

1. Provided python code
   The provided python code for a CLOCK (from `devices.py`) is provided below.

```
    def make_clock(self, device_id, clock_half_period):
        self.add_device(device_id, self.CLOCK)
        device = self.get_device(device_id)
        device.clock_half_period = clock_half_period

        # clock initialised to a random point in its cycle
        self.cold_startup()

```

   Effectively, the CLOCK is randomly initialised as 0 OR 1. Therefore, our implementation omitted an initial value for a CLOCK.

2. Simulating Real Clocks Due to physical (such as quantum) effects, clocks are usually impossible to synchronise. Therefore, there is no benefit to setting an initial bit for a clock.

The value in multiple clocks is having different speeds (set by period in our EBNF). This allows for communication with external devices (operating at a different clock speed) and optimise for performance (use higher clock speeds when rendering).

Note that we have assumed clocks are rising-edge triggered.

6.4 Forbidden States in DTYPE

```
    DEVICES C1:CLOCK (PERIOD:32),
            S1:SWITCH (INITIAL:1),
            S2:SWITCH (INITIAL:1),
            S3:SWITCH (INITIAL:1),
            D1:DTYPE;

    CONNECT C1 -> D1.CLK,
            S2 -> D1.DATA,
            S2 -> D1.SET,
            S2 -> D1.CLEAR;

    MONITOR D1.QBAR;
```

In this code, D1 has both SET and CLEAR to HIGH, resulting in a forbidden state. While physically, this should not be allowed in a DTYPE, the responsibility of handling this forbidden state has been passed over to the logic simulator. The programmer should still be allowed to define a forbidden state without error. Note that this is a design choice.

References

[1] *Logic Circuit Diagram Design*, Accessed on 21 May 2025, Available at https://https://online.visual-paradigm.com

[2] *Agile Team Planner*, Accessed on 23 May 2025, Available at https://www.figma.com