

20875 Software Engineering – Assignment 1

Due ~~Monday, November 10th~~ Friday, November 14th, 2025, 23:59 CEST

The assignment consists in implementing a simplified form of “placement” in integrated circuit design. Grading will be partially automated, and you are asked to follow precisely the instructions given in this document. In particular, the input, output, and exit status of the program must all conform to specific formats detailed below. The assignment is to be uploaded as a single archive file (either `.zip` or `.tgz`) on BlackBoard by the end of November 10th.

Rules. This is an individual assignment. You are allowed to talk about the assignment with your classmates. However, you must write all your code on your own. As a consequence, upon request, you will be able to explain and modify any part of your code.

Invocation. This assignment can be completed in Python or in C. For any other language, you must get approval first. The program will be invoked with the following commands:

Python: `python3 place.py {formulas} {image}`

C: `./place {formulas} {image}`

where `{formulas}` is the name of the input file containing Boolean formulas (e.g. “`circuit.txt`”), and `{image}` is the name of the output image file in the PPM format (e.g. “`circuit.ppm`”).

Exit code. The program exits with a zero exit code if the input file conforms to the specification. It exits with a nonzero exit code if it is invalid.

Compilation. [Only for assignments completed in the C language.] If a file called “`Makefile`” is present in the submitted archive, the code will be compiled with the command: `make`

Otherwise, it will be compiled with the command `clang -Wall -O3 -o place *.c`

External libraries/modules. You are allowed to use anything included in the standard library of the programming language you use. In addition you are allowed to use Python’s PIL module (Pillow), or the C libraries `libnetpbm` and `stb_image`. If you need anything else, you must get approval first.

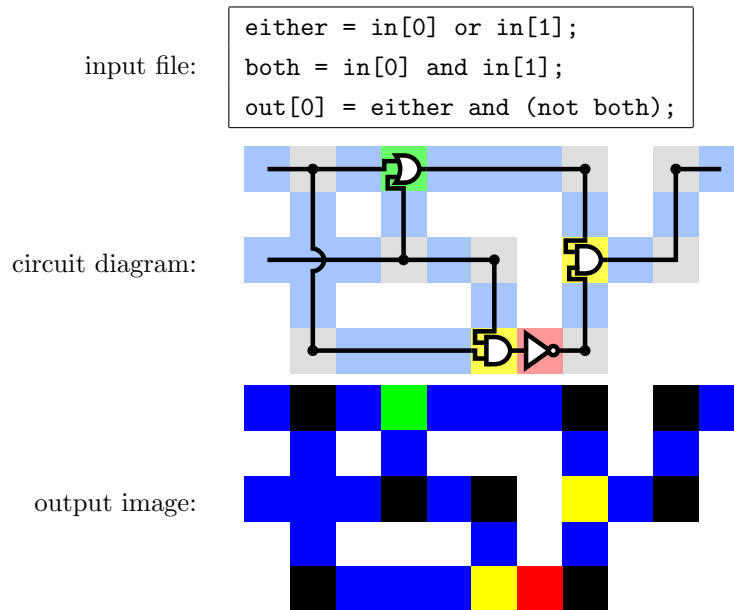
Grading.

- [3 marks] The program accepts all input files that conform to the specification and parses them correctly.
- [3 marks] The program correctly rejects malformed files (no crash or uncaught exception/error).
- [4 marks] For valid input files, the program produces a correct output image, as specified in this document. It exits (with a zero exit code) in 10 seconds or less for valid input files of size similar to the examples provided.
- [1 bonus mark] The program produces images with less than one million pixels for all input files tested.

1 Objective

The objective is for your program to read a set of Boolean expressions, and write an image whose pixels implement a circuit diagram corresponding to those Boolean expressions. For example, the following figure

shows the input and output of our program, where we want it to implement a XOR gate:



2 Input

The input must conform precisely to the following specification. Any program that cannot be tokenized or parsed according to these rules must be rejected with an error message. Crashes and uncaught exceptions are not considered appropriate error messages.

2.1 Tokenization

1. **Comments.** Anything on a line after the character “#” is ignored.
2. **Special characters.** There are 6 special characters: “(”, “)”, “[”, “]”, “=” and “;”.
3. **Numbers.** A *number* is any sequence of (one or more) consecutive *digits* (“0”, ..., “9”).
4. **Words.** A *word* is any sequence of (one or more) consecutive *letters* (“A”, ..., “Z” and “a”, ..., “z”), *digits* (“0”, ..., “9”) and *underscores* (“_”) that starts with a letter or an underscore.
5. **Blanks.** Spaces (“ ”), tabs (“\t”), carriage returns (“\r”) and newlines (“\n”) are considered *blank* characters. All blank characters are ignored except for the fact that they can separate two words/numbers.
6. **Keywords and identifiers.** There are 5 *keywords*: “in”, “out”, “not”, “and”, “or”. Any other word is an *identifier*.

2.2 Parsing

We define here how a program is to be parsed, once it has been tokenized into a sequence of numbers, keywords, identifiers and special characters.

A program is a sequence of *assignments*. Each *assignment* consists in (a) a *target*, followed by (b) the special character “=” and (c) an *expression*, then (d) a semicolon (“;”). A *target* (a) is either an identifier, or an output

denoted by $\text{out}[k]$ where j is a non-negative integer. It cannot have been previously assigned. Moreover, it cannot be used in its assigned expression, nor in any expression preceding its assignment.

The expression (c) is a Boolean formula involving *elements*. An *element* can be either a previously-assigned target, or an input denoted by $\text{in}[j]$ where j is a non-negative integer. An expression is either an element, or the negation (“not”) of a sub-expression, or a conjunction (“and”) of 2 or more sub-expressions, or a disjunction (“or”) of 2 or more sub-expressions. If those sub-expressions are not elements themselves, then they *must* be surrounded by parentheses. As a result, there is no need for operator priorities.

BNF grammar:

```





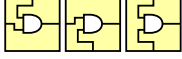

<target>      ::= <identifier> | "out" "[" <number> "]"
<element>     ::= <target> | "in" "[" <number> "]"
<paren-expr>  ::= <element> | "(" <expr> ")"
<negation>    ::= "not" <paren-expr>
<conjunction> ::= <paren-expr> "and" <paren-expr> | <paren-expr> "and" <conjunction>
<disjunction> ::= <paren-expr> "or" <paren-expr> | <paren-expr> "or" <disjunction>
<expr>       ::= <negation> | <conjunction> | <disjunction> | <paren-expr>

<assignment> ::= <target> "=" <expr> ";"
<program>    ::= <assignment> | <assignment> <program>

```

3 Output

The output image must conform precisely to the following specification. It is an image file in the PPM format. The width and height (w, h) of the output image is chosen by your program. Pixel coordinates are given by (x, y) where $(0, 0)$ is the top-left pixel and $(w - 1, h - 1)$ is the bottom-right pixel. Every pixel in the image has one of 6 distinct colors:

color	(red, green, blue)	operation	equivalent circuit
white	(255, 255, 255)	-	
black	(0, 0, 0)	connection (all sides)	
blue	(0, 0, 255)	connection (top-down, left-right)	
red	(255, 0, 0)	NOT	
yellow	(255, 255, 0)	AND	
green	(0, 255, 0)	OR	

Each pixel has 4 edges (top, bottom, left, right). Each edge can be an out-edge (whose Boolean value it drives) or an in-edge (Boolean value driven by an out-edge of the adjacent pixel). Every in-edge must be driven (either directly, or through black and blue pixel connections) by exactly one out-edge.

Pixels in the leftmost column ($x = 0$) on even lines correspond to the inputs. Specifically, pixel $(0, 2j)$ drives its right out-edge to the value $\text{in}[j]$ if $\text{in}[j]$ is used in any expression, and must be blue. Any other pixel in

column 0 must be white.

Pixels in the rightmost column ($x = w - 1$) on even lines correspond to the outputs. Specifically, pixel $(w - 1, 2j)$ must be driven by its left in-edge to the value of $\text{out}[k]$ if $\text{out}[k]$ is assigned, and must be blue. Any other pixel in column $w - 1$ must be white.

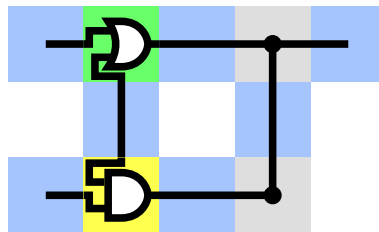
Pixels in the other columns ($1 \leq x \leq w - 2$) have the following meanings.

1. A white pixels does not connect anything. Pixels outside of the image behave like white pixels.
2. A black pixel connects its (up to four) neighbors together.
3. A blue pixel connects its top and bottom neighbors together (if present), and its left and right neighbors together (if present).
4. A red pixel must have exactly two neighboring pixels that are not white, one being to its right. It drives its out-edge, on the right side, to the negation (NOT) of its in-edge (driven by the other neighbor).
5. A yellow pixel must have exactly three neighboring pixels that are not white, one being to its right. It drive its out-edge, on the right side, to the conjunction (AND) of its two in-edges (driven by the other neighbors).
6. A green pixel must have exactly three neighboring pixels that are not white, one being to its right. It drives its out-edge, on the right side, to the disjunction (OR) of its two in-edges (driven by the other neighbors).

Given those rules, the image must be such that the value of the outputs ($\text{out}[k]$ for all k) is driven to the appropriate Boolean function of the inputs ($\text{in}[j]$ for all j), as described by the input file.

Beware that not every image composed of the six colors above is a valid circuit. In particular, two out-edges can never be connected together (neither directly, nor through black or blue pixels). For example, the following circuit is invalid in three ways:

- The bottom in-edge of the OR gate at $(1, 0)$ is not driven,
- the top in-edge of the AND gate at $(1, 2)$ is not driven, and
- the in-edge of $\text{out}[1]$ at $(4, 0)$ is driven by two out-edges.



Invalid circuit

4 Example

The following example gives an input file and one possible output image for a ‘full adder’. Both can be downloaded using these links:

`full_adder.txt`

`full_adder.ppm`

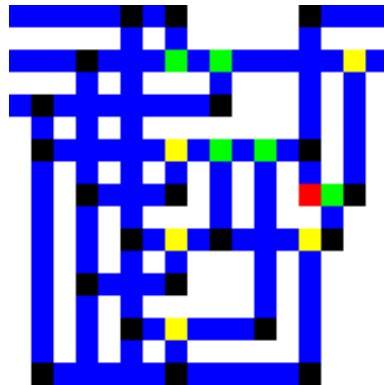
input file:

```
Cin = in[0];
A = in[1];
B = in[2];

Cout = (A and B) or (A and Cin) or (B and Cin);
S = ((not Cout) or (A and B and Cin)) and (A or B or Cin);

out[0] = Cout;
out[1] = S;
```

output image:



5 Checklist

A few quick checks to prevent common and easily avoidable mistakes:

- Assignment uploaded as a single `.zip/.tgz`
- [Python] Main script is called `place.py`
- Code takes the correct two arguments as specified (input file and output file)
- [Python] Script does not abort with uncaught exceptions under any circumstance
- [C] Executable does not crash even when the input is invalid (test a few invalid files)
- When the input file is valid, the code properly parses it and the exit code is 0
- When the input file is invalid, the code rejects it and the exit code is not zero