02332 Compiler Construction
Mandatory Assignment 1: A Simple Hardware Simulator

Hand-out: 12. September 2023
Due: 13. October 2023
Hand-in via Learn platform in groups of 3-4 people
To hand in:

- All relevant source files (grammar and java)
- Your test examples.
- A small (1-2 page) report in PDF format that documents what you did for each task (including answers to questions of the task), possibly with code snippets.

HDL0: A Simple Hardware Description Language

We define a language HDL0 for describing hardware circuits. This first assignment will implement a lexer and parser for HDL0 using ANTLR and an output in HTML format. This output can then be viewed with a normal web browser.

The second assignment (after the fall holidays) will then be to implement an interpreter for HDL0, simulating a given hardware circuit for a given input. For the second assignment you will get a model solution of this first assignment.

HDL0 is inspired by existing hardware description languages, but a bit simplified and reduced to a few key concepts.

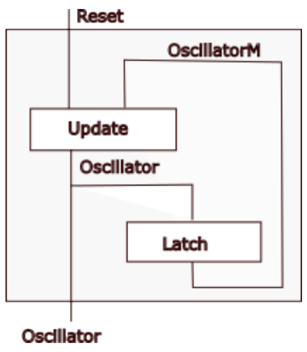
We have in principle three kinds of things in HDL0:

- *Signals* (or wires) that at any time point have either a high value ("1") or a low value ("0").
- A *combinatoric* part: computing using logical functions on signals like conjunction (logical "and", written `&&`), disjunction ("or", written `||`), and negation ("not", written `!`).
- Latches, i.e., memory cells that are connected to one global clock, and at every clock tick read the current value of an incoming signal and output that until the next clock tick.

Example

To make the explanation concrete, let us look at a simple example first (found in the file `01-hello-world.hw`). Note that the following text also explains on this example how HDL0 works, i.e., what this HDL0 specification actually means. For solving this first assignment it is actually not strictly necessary to understand the meaning of HDL0 (it will be only really relevant for the second assignment when you shall write an interpreter). However, it usually helps to have an intuitive understanding of the meaning of a language to begin with.

```
.hardware helloworld
.inputs Reset
.outputs Oscillator
.latches
Oscillator -> OscillatorM
.update
Oscillator = !OscillatorM && !Reset
.simulate
Reset=0000100
```



The diagram on the right is just for illustration in the style hardware circuits may be depicted; here the convention is that inputs come in the top of a component and outputs come out at the bottom. The circuit has one input signal `Reset` and one output signal `Oscillator`. The update block is the combinatoric part of the circuit and it computes from `Reset` and `OscillatorM` the signal `Oscillator`.

`Oscillator` is both the output of the circuit and the input to the latch – a one bit memory that has output `OscillatorM`.

The idea is that the latch is connected to a clock (not shown in the diagram) and at each tick of the clock it reads the input `Oscillator` signal, stores it, and outputs it on the output `OscillatorM` until the next clock tick. We assume that the time between two clock ticks is larger than the time that the update circuit needs to compute its outputs, so, effectively, `OscillatorM` is the value of `Oscillator` of the previous clock cycle.

The update block is now a combinatoric circuit: it computes `Oscillator` as `!OscillatorM && !Reset`, i.e., computing the *not* of the inputs and then the *and* of that. Thus, as long as the `Reset` is 0, `Oscillator` will be the negation of `OscillatorM`, thus, at every clock cycle, `Oscillator` will switch its value, hence the name. If the `Reset` is however 1, then the `Oscillator` will be 0 in the following clock cycle. (We assume here that inputs are stable over a clock cycle to keep matters simple.)

An interesting question is: what is the initial value of `OscillatorM`? In fact, in reality non-deterministically either 0 or 1, but for simplicity, we just define that in HDL0, all latches in the initial clock cycle output the value 0.

Finally, there is a `.simulate` section that gives a bitstring for every input signal; this bitstring means the value that the input signal should have during each clock cycle, so in the example, the `Reset` signal would only be set in the fifth cycle of the system.

The task of the simulator *in the next assignment* will be to compute the corresponding output signals also as such a bitstring. For this example we would have:

```
0000100 Reset
1010010 Oscillator
```

where we can see that the Oscillator is alternating between 0 and 1 until the point where the `Reset` forces it to stay at 0. The first value of the `Oscillator` is 1, because the initial value of `OscillatorM` is 0 and reset is not set in the first cycle.

Syntax of HDL0

Generalizing from the concrete example, here is an informal description of the syntax. An HDL0 specification consists of the following sections (in the given order):

- `.hardware` just defining the name of the circuit
- `.inputs` specifying a list of input signals like `A B C`. We require that there is at least one input signal.
- `.outputs` specifying a list of output signals like `A B C`. We require that there is at least one output signal.
- `.latches` specifying a list of one-bit memory cells in the form `In -> Out`. One should be able to specify any number of such latches.
- `.update`, the combinatoric part of the circuit. The update consists of updates of the form `Out = Exp` where `Out` is a signal and `Exp` is an expression over signals and the Boolean connectives *and*, *or*, and *not*. There can be any number of updates.
- `.simulate` consists of input specifications of the form `In=Values` where `In` is an input signal of the circuit, and `Values` is a sequence of Booleans. This specifies the values that this input signal should have for some number of clock cycles.

We also allow comments like in Java or C++ (single line comments between `//` and newline, as well as multi-line comments between `/*` and `*/`). Also whitespaces (space, tab, return) are not significant.

Task 1 (Week 3/4)

The first task is to understand all that has been said up to here :-) and to define a context-free grammar for HDL0 in ANTLR. You should test this grammar, in particular to accept all three provided examples.

We suggest to first try to get the example from the lecture in week 3 to run with ANTLR and Java. Please reach out to the teachers, if you have still issues to run it. You may well use this source code and Makefile as a starting point for this project, since several things can be just re-used with a bit of adaptation.

A good strategy is to first focus on the `.update` section, i.e., the combinatoric part of the circuit. One can design an ANTLR grammar for just this part of the language and then test it on the respective part of the given examples.

Once this works satisfactorily,[1] you can proceed to specify a grammar for the full specification. Again check with experiments that the grammar works on the given examples.

It is a good idea to make a "lab protocol" of your experiments so you can easily write a report explaining your design choices and document that you have properly tested the implementation.

Task 2 (Week 4/5)

Check what ambiguities your grammar contains and whether they are resolved in the preferred way. In particular, negation binds stronger than conjunction, and conjunction binds stronger than disjunction. For instance

`A || !B && C` should mean `A || ((!B) && C)`

Check that this is parsed correctly on all examples, and check if there could be other ambiguities that are not covered by the examples.

Again, a lab protocol can be very helpful for later writing your report.

Task 3 (Week 5/6/7)

This week is about implementing a visitor that we call `prettyprint`. The task of `prettyprint` is to generate output in HTML format of the given circuit. This does not include much computation, but is just an exercise to work with parse trees and to adapt output to a given language like HTML. In fact this HTML output will be very close to the input file, with only a few transformations. The output of your compiler should be tested using a web browser.

There is a bit of "boilerplate" code that should be generated to load the MathJax javascript package:

```
<!DOCTYPE html>
<html><head><title>TITLEOFTHEPAGE</title>
<script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
<script type="text/javascript" id="MathJax-script"
async src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-chtml.js">
</script></head><body>
THEMAINTEXT
</body></html>
```

where `TITLEOFTHEPAGE` and `THEMAINTEXT` should be replaced by something meaningful.

The `prettyprinter` should...

- ... put the name of the specification into `<H1>...</H1>` (big headings) and then have smaller headings (with `H2`) for sections **Inputs**, **Outputs**, **Latches**, **Updates** and **Simulation inputs**.
- The sections inputs and outputs should just list the respective signals
- The latches section should list all the latches in the form *inputsignal* → *outputsignal* (the rightarrow in HTML is `&rarr;`)

[1]During these experiments it may turn out that some `.update` sections are accepted by the parser, but the parse tree does not match your expectations. This is part of Task 2 below and can be ignored for now.

- The updates specification should have the form *outputsignal* ← *expression* (the leftarrow in HTML is `&larr;`)
- The expression itself should be set in LaTeX:
  - it needs to be surrounded by parenthesis with a backslash to signal, like so: `\( latexcode \)`
  - for logical *and* we use `\wedge`
  - for logical *or* we use `\vee`
  - for logical *not* we use `\neg`
  - signal names in latex should be inside a `\mathrm{...}`
  - To avoid ambiguities in the output, every expression should be surrounded by parentheses.
- Also helpful for printing really pretty: in HTML, one can enforce a line-break with `<br>`.

Note that in order to produce strings in Java, you need to make some *escapings*:

- To generate a line-break you need to have `\n`
- For `\` you need to write `\\`
- For `"` you need to write `\"`
- ...

For the simple helloworld example, the output could look like this:

```
<!DOCTYPE html>
<html><head><title> helloworld</title>
<script src="https://polyfill.io/v3/polyfill.min.js?features=es6"></script>
<script type="text/javascript" id="MathJax-script"
async src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-chtml.js">
</script></head><body>

<h1>helloworld</h1>

<h2> Inputs </h2>
Reset

<h2> Outputs </h2>
Oscillator

<h2> Latches </h2>
Oscillator&rarr;OscillatorM<br>

<h2> Updates </h2>
Oscillator&larr;\((\neg(OscillatorM)\wedge\neg(Reset))\)<br>

 <h2> Simulation inputs </h2>
<b>Reset</b>: 0000100<br>

</body></html>
```