02332 Compiler Construction
Mandatory Assignment 1: A Simple Hardware Simulator

Hand-out: 17. September 2024
Due: 11. October 2024
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.

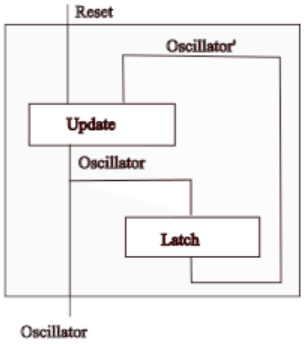
HDL0: A Simple Hardware Description Language

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.

Example For this assignment its actually not utterly necessary to understand what HDL0 means — we are just concerned with syntax for now — but it usually helps to get an intuitive understanding first. So here is a simple example (found in the file `01-hello-world.hw`) and for illustration a diagram in the common style of depicting hardware circuits to get a visual impression.

```
hardware: helloworld
inputs: Reset
outputs: Oscillator
latches: Oscillator
updates:
Oscillator = /Oscillator' * /Reset
siminputs:
Reset=0000100
```



First, the description has the name `helloworld` but this name has no further significance. We have one input signal `Reset`. All signals have at any time point have either the value "0" (Low) or "1" (High), and their value can change over time. The fact that `Reset` is an input signal means that the value is given by the environment, for instance there could be a reset button that a human user can push, and while this button is pushed, the `Reset` signal is "1", and otherwise it is "0".

Now let us look first at the `updates:` here we have the line

```
Oscillator = /Oscillator' * /Reset
```

This means that here we compute a signal `Oscillator` from signals `Oscillator'` and `Reset`. `Reset` is the input we just discussed, the `Oscillator'` is an internal signal of the circuit (in fact, coming out of the latch as explained below). First, the slash symbol "`/`" means *negation* ("not"). If `Reset` is "0" then `/Reset` is "1" and vice-versa. The `*` is *conjunction* ("and"). Thus, the signal `Oscillator` is "1" if and only if both the signals `Oscillator'` and `Reset` are "0".

Next, the `Oscillator` signal is both the output of the circuit and input to a *latch*, the simplest for of a memory cell for a single bit. We assume that we have a global clock to which all latches are connected. At each clock tick the latch writes the current value of the incoming signal `Oscillator` into its memory cell. Every latch has an output signal; to make the name of output signal simple, in our

hardware language, we just append a prime symbol (') to the name of the input signal. So `Oscillator'` is the output signal of the latch that reads `Oscillator` at every clock tick. Thus, `Oscillator'` saves the value that `Oscillator` had at the last clock tick. This allows to use this value to be used in the update to compute the next value of the oscillator. In fact, assume that `Reset` is permanently "0", the Oscillator would at each clock tick invert its change its value. When the `Reset` button is pressed, then the Oscillator is set to value "0" for the current clock cycle.

As an example "run" of this circuit, consider the given `siminputs`: this gives a concrete value for the input `Reset` where each digit should mean the value for one clock cycle. Then we get the following *simulation*:

Timepoint	1	2	3	4	5	6	7
`Reset`	0	0	0	0	1	0	0
`Oscillator`	1	0	1	0	0	1	0
`Osclillator'`	0	1	0	1	0	0	1

First observe that `Oscillator'` is just the signal `Oscillator` delayed by one clock cycle. That is essentially the behavior of the simple latches that we use here. One may wonder why the initial value of `Oscillator'` is "0". This is actually a common problem in hardware design: the initial value of memory cells is in general non-deterministic. We for simplicity assume in our language that all memory cells are initially "0". Now you can check that `Oscillator` indeed is correctly computed from `Oscillator'` and `Reset` according to the update line.

To summarize, the hardware circuit consists of basically two parts:

- A combinatoric part (the `updates` section) that has no memory cells and no cycles and that just computes outputs as a function of some inputs using logical functions like "and" and "not".
- The latches that allow us to store values and to have "cycles" in the circuit, in the example that the output of a latch is input to the combinatoric part.

Definitions One additional concept that HDL0 has to make the life of hardware designers easier: defining small combinatoric circuits with one output. For instance we could actually rewrite the circuit as:

```
hardware: helloworld
inputs: Reset
outputs: Oscillator
latches: Oscillator
def: nor(a,b)= /a * /b
updates:
Oscillator = nor(Oscillator',Reset)
siminputs:
Reset=0000100
```

Here we have defined a circuit `nor` that has two inputs `a` and `b` and computes the function `/a*/b` (which actually is true if neither `a` nor `b`). We can use it in the computation of the `Oscillator`. This does not make the description shorter (maybe slightly easier to read), but for larger circuits it can really help to define oneself such abbreviations. See for instance the example `04-vonNeumann.hw`.

Syntax of HDL0

Generalizing from the concrete example, here is an semi-formal description of the syntax. First, an *expression* is any of the following:

- A signal like `Reset`
- The conjunction ("and") of two expressions `Exp1 * Exp2` where one may even omit the symbol star. For instance instead of `A*B` one may just write `A B`.
- The disjunction ("or") of two expressions `Exp1 + Exp2`

- The negation ("not") of an expression `/Exp`
- The use of a definition of the form `f(EXPS)` where `f` is an identifier and `EXPS` is a comma-separated list of expressions.[1]
- An expression in parentheses `(EXP)` is also an expression.

The priority is that `/` binds strongest, `*` second strongest, and `+` binds weakest. For instance `/A+/B*C` is equivalent to `(/A)+((/B)*C)`

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 signals that are fed into one-bit memory latches. (For an input signal `A`, the output signal is called `A'`, so identifiers that end on a prime symbol should be allowed.) There can be any number of latches.
- Optional definitions of the form `def: f(Args) = Exp` where `f` is an identifier, `Args` is a comma-separated list of signals, and `Exp` is an expression. There can be any number of definitions.
- `updates:`, 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. There can be any number of updates.
- `siminputs:` 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 that the ANTLR-generated parser accepts all provided examples.

A good strategy is to first focus on the `updates:` section, i.e., the combinatoric part of the circuit. One can start with designing an ANTLR grammar for just this part of the language and then test it on the respective part of the given examples. You may use and adapt examples from the lecture for handling the updates.

Once this works satisfactorily,[2] 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 as already mentioned. 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.

[1] It is part of later analysis (in the second assignment) to check that there really is a def of `f` with the correct number of arguments; here we can just accept any identifier `f` with any number of arguments.

[2] During these experiments it may turn out that, due to ambiguities, the parse tree has a different shape than the one you would like. Handling this is part of the second task of this assignment for week 4/5.

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**, **Definitions**, **Updates** and **Simulation inputs**.
- The sections inputs and outputs should just list the respective signals
- 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.
- When a definition like `nor` in the example is used in an expression, please set the identifier in italics using latex command `mathit{...}` to distinguish from signals.
- 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 (with the `nor` definition), 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
<h2> Definitions </h2>
\(\mathit{nor}(a,b)=(\neg(\mathrm{a})\wedge\neg(\mathrm{b}))\)<br>
<h2> Updates </h2>
Oscillator&larr;\(\mathit{nor}(\mathrm{Oscillator'},\mathrm{Reset})\)<br>
<h2> Simulation inputs </h2>
<b>Reset</b>: 0000100<br>
</body></html>
```