02332 Compiler Construction
Mandatory Assignment 2: A Simple Hardware Simulator

Hand-out: 22. October 2024
Due: 22. November 2024 23:59
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.

The Hardware Description Language HDL

The first assignment was designing the "front-end" of the simulator for the small hardware description language HDL – the lexer and parser, as well as a HTML output. On DTU Learn, a solution to this first assignment is found and that is suggested as a starting point for you for the second assignment.

This solution in fact includes a bit more:

- The file `AST.java` contains data structures for an abstract syntax for HDL. We highly recommend to use this file for the second assignment, i.e., to implement the simulator mainly by extending `AST.java` with new methods that perform checks and run the simulation.
- The file `main.java` contains two visitors:
  - The visitor `JaxMaker` produces the HTML output – solving what was required in the first assignment.
  - The visitor `ASTMaker` produces the abstract syntax in the data structures of `AST.java`. More precisely for the start symbol of the grammar, it generates an object of class `Circuit` that contains all the data of a hardware specification (inputs, outputs, latches, updates, and the simulation inputs).

  For this second assignment, you can ignore the `JaxMaker`, and just use `ASTMaker` to get abstract syntax. You do not have to modify this visitor, and you can entirely work with the AST. (No more visitors!)
- The file `Environment.java` is an adaption of the file we used for the interpreter in the lecture, just here it maps to `Boolean`. The idea is that during simulation, at every time point, a signal has a Boolean value. Additionally, the environment contains a list of all function definitions of the circuit (the "def"), so that an interpreter can "look them up" easily. This can be used in task 2 of this assignment.

Task 1 (Week 6/7)

This task is to implement the core of the simulator. We suggest the following design:

- Implement a method `eval` for all expressions. This method should take as argument an Environment, so it can look up the current value of each signal in the circuit. As output, it should give a Boolean for the computed expression.
  - If the expression contains a signal that is not defined in the environment, it should stop with an error.
  - For the class `UseDef`, i.e., using of user-defined functions, stop here with an error message for now; this shall be implemented in task 2 of this assignment.

- Note that the class `Update` represents one line in the `update` section of the form `signal=expression`. Write for this class a method `eval` that sets the value of the defined signal to the value that the given expression currently yields. This method `eval` also takes an Environment as argument, but returns nothing.

- For handling latches, first recall that we declare a number of signals as latches (the list `latches` in class `Circuit`); note that each latch signal is the input to a latch, and the output is the same signal with a prime (') added to it. So for instance declaring `latches A,B,C` means we have three latches with inputs `A,B,C` and outputs `A',B',C'`. At the start of the simulation, all the outputs (like `A',B',C'`) should be initialized to 0:

  - Write a method `latchesInit` of class `Circuit` that takes an environment as argument and sets all latch outputs to value 0 in this environment.
  - Write a method `latchesUpdate` of class `Circuit` that also takes an environment as argument and sets every latch output to the current value of the latch input. In the example above, it would write to `A'` the current value of `A`, and similar for `B` and `C`.

- The class `Trace` represents the sequence of values that signal takes over time. Write a method `public String toString()` to produce the output for that trace.

- For class `Circuit`: implement a method `initialize` that again gets an Environment as argument and should do the following:

  - Read the input value of every input signal at time point 0 from the siminputs and make an entry into the Environment. This thus initializes all input signals. This method stops with an error if the siminput is not defined for any input signal, or its array has length 0.
  - Call the `latchesInit` method to initialize all outputs of latches.
  - Run the `eval` method of every `Update` to initialize all remaining signals.
  - Print the environment on the screen (note it has a `toString` method), so one can see the value of all variables.

- Also for class `Circuit`: implement a method `nextCycle` that takes as an input both an Environment and an `int` i that represents the cycle number the simulation is at (the previous method initialize is cycle number 0). It should do the following:

  - It should read the input value of every input signal at time point i from the siminputs and make an entry into the environment. Again, this errors if the i-th entry in the siminput is not defined.
  - Call the `latchesUpdate` method to update the output signals of latches.
  - Run the `eval` method of every `Update` to update all other signals.
  - Again print the environment.

- Write a method `runSimulator` that runs `initialize` and then n times `nextCycle` where n is the length of the simulator inputs. You may here assume that all siminputs have the same length.

Test that your simulator works for the example input files that do not have definitions, i.e., check that the outputs of the simulator match what it should be.

Task 2 (Week 7/8)

Now we take care of user-defined functions. For example if the hardware file contains the definition:

```
def xor(A,B) = A * /B + /A * B
```

then expressions can use it, for instance `X + xor(X+Y,Z)` is now also an expression.

One could handle this by a compilation step that simply replaces every occurrence of `xor` in the expressions by its definition, i.e., it would replace the expression `X + xor(X+Y,Z)` by

```
X + ((X+Y) * /Z + /(X+Y)* Z)
```

This is however a bit tricky to implement, and we do not recommend it.

The easier way is to just handle this like a function call in a programming language. We describe this at hand of the above example `xor(X+Y,Z)`:

1. We create a new environment (check below: there is a special constructor for this situation).
2. For each formal argument (`A` and `B` in the example) we need to compute the actual argument (the value of `X+Y` and `Z` in the example). This computation is within the current environment. Say it gives `1` for `X+Y` and `0` for `Z`. Then we should have as a new environment `A` set to `1` and `B` set to `0`.
3. With this new environment, we evaluate the body of the definition, i.e., `A * /B + /A * B`. This gives again a Boolean (in the example `1`) and that is the result of this function call.

The difficulty in implementing this is that one needs to look up the definition of function like `xor` during the `eval` method. For this reason, the given implementation of `Environment` is extended to include an additional map `defs`; it maps from function names like `xor` to their definition. The relevant functions are:

- `public Environment(List<Def> listdefs)` – this is a new constructor one can use to create an initial environment: give the list of definitions of the `Circuit` as argument to the environment.
- `public Def getDef(String name)` – this function can now be used to obtain the definition of a given function, e.g., during the evaluation of an expression calling `xor`, one can use `getDef("xor")` to obtain the definition.
- `public Environment(Environment env)` – this function can be used to create a new Environment and initializing the definitions with those of an existing one. This is needed when implementing the `eval` method, because one can create the new environment with just all the function definitions of the current environment.

Task 3 (Week 8/9/10)

In the simulator from the previous weeks, the output is a bit hard to read, because we get the "snapshot" of all signals at every clock cycle. Instead we want now that after the simulation is over, we can see the trace for each input and output signal over the entire simulation. For instance for the `01-hello-world.hw` example, the output would look like this:

```
0000100 Reset
1010010 Oscillator
```

In fact, it is suggested to write the Booleans first and the name of the signal second, so it is nicely aligned even if the signal names are of different lengths. Hint: use the member variable `simoutputs` of `Circuit`.

Week 11

Lab week to complete the implementation and report.