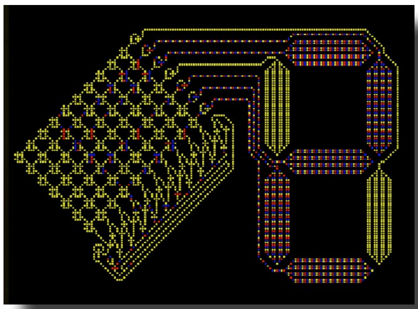
Australian
National
University

WIRING UP WIREWORLD

First assignment for Introduction to Programming and Algorithms 2014

Uwe R. Zimmer based on work by Robert Offner (Probie)

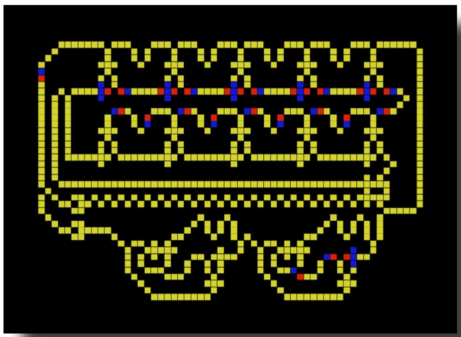


What are you looking at on the right?

- A seven segment display showing the number 2 ?
- A picture of a seven segment display showing the number 2 ?
- A picture of a computer emulating electronic circuits?
- A machine emulating a machine emulating a machine?
- A weird bunch of colour spots?
- A very simple machine, yet as powerful as any other computer?
- A pattern which is probably supposed to change over time?

Yes, you guessed right: this is a **cellular automaton**. And pattern you see (the whole picture) is its current **state**. Most cellular automata have a state which is a finite, fixed (here: two) dimensional, rectangular pattern (grid, matrix) of **cells** – which makes it easy to display on a screen. Each cell is in exactly one out of finite list of discrete **cell states** at any time. As these cell states are finite and fixed, we give each cell state a colour and just print a dot of this colour for each cell in the automaton. So far we only have a weird way to describe a bitmap picture.

The interesting aspect of the cellular automata lies in the rules which are used to progress from one state to the next. So the pictures on this page are actually only snapshots of states and the cellular automata will make them move. This is done by a (usually stunningly small) set of rules which only take the current state of the cell itself and its local neighbourhood into account.

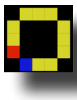


The specific cellular automaton which you see here is called **Wireworld** and is defined by four possible states for each cell:

- ⬛ *Empty*
- 🟨 *Conductor*
- 🟦 *Electron head*
- 🟥 *Electron tail*

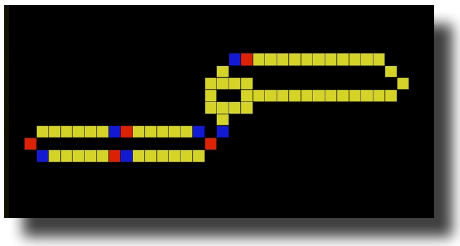
and the accompanying four rules to progress each cell into the next state:

- ⬛ *Empty* ⇒ ⬛ *Empty*
- 🟨 *Conductor* ⇒ {
  - 🟦 *Electron head* ; if 1 or 2 of the 8 adjacent cells are *Electron heads*
  - 🟨 *Conductor* ; otherwise
- 🟦 *Electron head* ⇒ 🟥 *Electron tail*
- 🟥 *Electron tail* ⇒ 🟨 *Conductor*



This sounds like it is just meant to send an electron through a wire, and in fact it does that too. Look at the simple state on the right and progress it a few states further by applying the rules of Wireworld (in your head or on a piece of paper).

That's all? All this effort just to move an electron in a circle? It turns out that this is just the beginning and those four cell states and rules of Wireworld can be employed to express *any* form of complex computation which for example your computer in front of you is capable of. To get into the mood, have a closer look at this state to the right and progress it a few states further as well in your head (or on a piece of paper).



This state turns out to implement the functionality of an Exclusive-Or gate (the logical function which returns true if exactly one of the inputs is true). Any other logic gate can be formulated in a similar way, and based on what we know about logic: we can build any digital computer out of basic logic gates[1].

While Wireworlds can implement logic gates, they are more flexible and we do not need to restrict ourselves to a design in logic gates alone. As you will watch the Wireworlds move along in a few days, you will find that there is for instance also a quite complex encoding of time and synchronization happening. Cellular Automata have been frequently discussed as an alternative mode of computation and also have interesting relations to **Quantum Computing** and **Massively Parallel Computing**. I'll leave it to you to dig deeper, if you are so inclined, but here we continue with your concrete assignment goal.

Programming task

Your task is to make the Wireworld move. In Haskell terms you will need to implement the function:

```
transition_world :: World_model -> World_model
```

In fact your will implement it twice:

```
transition_world :: List_2D          Cell -> List_2D          Cell
transition_world :: Ordered_Lists_2D Cell -> Ordered_Lists_2D Cell
```

based on two different underlying data structures used to store the current world (state of the automaton) while the `Cell` states are simply:

```
data Cell = Head | Tail | Conductor | Empty.
```

In the first case the world is stored as a single list where each cell has accompanying coordinates attached. Thus `List_2D` translates into

```
type List_2D e = [(e, (X_Coord, Y_Coord))]
```

where `e` is a placeholder for any type (we use it for our type `Cell`). No particular order on those coordinates is assumed.

In the second case the world is split up into lines (cells sharing the same y coordinate) and each line is stored in a separate list. Each line has a *y* coordinate, and each cell inside the line has an *x* coordinate attached to it. The whole world is then a list of those lines. Thus Ordered_Lists_2D breaks down into:

```
type Ordered_Lists_2D e = [Sparse_Line e]

data Sparse_Line e = Sparse_Line {y_pos :: Y_Coord, entries :: Placed_Elements e}

type Placed_Elements e = [Placed_Element e]
data Placed_Element  e = Placed_Element {x_pos :: X_Coord, entry :: e}
```

where `e` is again a placeholder for any type. All `Sparse_Line`s are ordered ascending by `Y_Coord` and all cells inside `Placed_Elements` are ordered ascending by `X_Coord`.

1 In fact we have too much already, as a "not-and" or NAnd gate alone is sufficient to implement any other part of your computer.

The two `transition_world` functions which you need to complete are found in the two modules in the directory `Sources/Transitions`.

As both data structures offer the same functions it would be possible to write the same code for both of your `transition_world` functions – yet think carefully whether this would be a clever idea. While the two data-structures offer the same logical functionality, the runtimes which they take to do so may be quite different. Hence it will make sense to write your transition functions in a way that they will exploit the characteristics of the underlying data structures. If you are unsure how they data structures differ, you may also start by running the same code in both functions and measure the number of transitions which each of them will achieve per second (read on to find out how to measure those). This will give you a hint as to what's happening here. Then start exploring how you could re-arrange your code in order to benefit from the characteristics of the two offered data structures.

Note that none of those data-structures is "dense", i.e. not all coordinates need to be explicitly represented. As you will have noticed when looking at the rules: the *Empty* cell in Wireworld will always stay *Empty*. As it will never change anyway, we don't need to store it. Similar to the concept of **negative space** in art, it is still part of the world, while we will never actually handle *Empty* cells. Those data structures are also not limited in size at any time. Wireworlds in its standard definition will not take advantage of this feature though, as they do not grow or shrink – all action happens on the existing wires. It is still a handy feature for some other cellular automata where cells can "spawn" or "vanish" into empty space and this "growth" or "shrinkage" is not necessarily limited either.

Setting up your account or computer for the assignment

Firstly, if you have not already done so, download the source code for the assignment from the course web-site. Secondly (if you want to run the assignment also on your own computer), install the gloss and bitmap libraries for Haskell. This can be done by running the following. Those libraries are already installed in the labs.

```
> cabal update
> cabal install gloss bmp
```

from the terminal (or command prompt on windows). You can check if it works by going to the top directory of the Wireworld sources (where you find `make_Wireworld`) and compiling it with[2] (in one line) (no need to type it: the line is also given to you in the file `make_Wireworld`, which you can run with "`./make_Wireworld`"):

```
 > ghc -O2 -W Sources/Wireworld.hs -iSources
-odir `uname -m` -hidir `uname -m` -o Wireworld
```

You can now run your freshly built program by "`./Wireworld`", but as you didn't write any transition function yet, the world will remain static. You can drag the world with the mouse, right-click-drag to rotate the world, or use page-up/page-down (or mouse scroll wheel) to magnify it. If this all works, you are technically set to begin the assignment, yet you need a concept first.

How to design your functions

You are now confronted (and quite likely for the first time) with a program design job which clearly goes beyond "fill in the gaps in the way we did in the last lecture". The provided frame-

2 "`--make`" means that the compiler checks what has been changed lately and only re-compiles the new files and files which depend on them – "`-O2`" stands for "`O`ptimize level `2`" (basically telling the compiler to put some effort into it) – "`-W`" activates all `W`arnings (which gives you more hints about things which might be wrong) – "`iSources`" tells the compiler where the source files are – "``-odir `uname -m` -hidir `uname -m` ``" tells the compiler to keep your source directories neat and tiny and store the object and interface files in a directory which has the name of your CPU architecture – "`-o Wireworld`" specifies the name of the `o`bject (executable) file).

work is specifically designed to be neither suggestive nor unnecessary restrictive, thus there are hundreds of different ways how to approach the problem. While this sounds good in principle, you might very well find yourself staring at the cursor for a while and nothing is coming up.

While you can of course design your functions in any way you see fit, I suggest the following basic process to get there:

a. *Grasp the problem:* What is the overall goal? What needs to happen to get there?
b. *Understand the restrictions and possibilities of the two given data structures:* If the world is represented in either of those two ways: Which forms of access to those data structures comes easy and natural, and which forms of access are clunky? The provided access routines look identical (and actual provide the exact same logical functionality), but they will work at quite different levels of efficiency – you can reason this out on a piece of paper or you can have a look at the two modules for the basic data structures and see how those access routines are implemented. The number of provided access functions is a hopeless overkill for the task at hand. You will only need very few of those in the end, yet understanding more of them gives you more options. Looking over them might also give you ideas for good concepts in the next step. Use only those functions which you fully understood.
c. *Come up with two concepts* of how to attack the problem with either of those two data-structures as the underlying model. If you understood the data-structures, your two approaches will most likely be different. All programming designs need to be efficient and elegant (I deliberately leave this vague for the moment). For instance, throwing everything in the air and check whether things are coincidentally sorted once they are back on the ground, is probably not an overly efficient and elegant concept for sorting. Some concepts will be *a lot* faster than others and while **speed** is usually not the only concern when designing a new concept, it is *always* one of them (common other concerns are **logical correctness** and **maintainability**). In concrete terms for your problem at hand: Consider that your program will be used for much larger worlds – will it still hold up?
d. *Break it down*: You will most likely not want to implement the transition as one big, monolithic monster function, but you will want to break it down into functions which are easy to understand and implement. This breakdown will determine the structure of your final program. For example you might think how to transition a single cell before you transition the whole world. Or you might want to factor out functions which will be required frequently, like counting the number of a specific cell type in the local neighbours. Those are just ideas and you can break down your program in many different ways. *Hint*: spend some time to come up with really good names when you break down your functions. While it seems tempting for new programmers to "get it over with" and slap down code massacre lines like "`helper3 = ma (test temp2) i worry`", you will not only drive your tutor mad (and thus lose marks), but you will also lose sight of what you are doing very quickly yourself.

Hints

If you already have a plan on what to do at this point, then just skip this whole section. Otherwise you might find a few valuable hints in here. There is absolutely no need to follow any of them, but some might make the start of your assignment somewhat easier.

We already hinted above that a general design method is to "break down" the problem. Now how does this work in praxis? Say your "main" problem is to implement:

```
transition_world :: List_2D Cell -> List_2D Cell
```

In-order-to-do-this you first might need to be able to transition a single cell. So maybe you want to write such a function first :

```
transtion_cell :: Element_w_Coord Cell -> List_2D Cell -> Cell
```

In-order-to-do-this you might need a function first which extracts the local neighbourhood around the cell at hand:

```
local_elements :: Coord -> List_2D e -> List_2D e
```

and a function which can count the number of a certain kind of cell in this neighbourhood:

```
occurrence_of :: Cell -> List_2D e -> Int
```

You can also directly reflect such a problem break-down in your code by using a `where` clause whenever you used the phrase “in order to do this”. In fact you should use scoping features (language features which allow you to limit visibility of code sections – like a `where` clause), if the smaller function is only used in this context and the only reason it exists is to be able to implement this specific larger (surrounding) function.

At some point you will obviously need to implement the actual rules of Wireworld. You will probably find that those rules translate completely naturally into a `case` expression.

But where do you actually start? Experienced programmers can see what’s coming and often implement the functions starting from the most complex one and then fill in the smaller functions. Especially for beginners it seems easier though to start with the simple-most functions first (which can also be implemented and tested independently). So if you find yourself in confusion, maybe start by writing a function which tests whether a certain cell is a head. Then continue with a function which counts the number of heads in a given list, etc. All those little (and slowly growing larger) functions can be tested on their own and you know that you are making progress.

Features of the provided code

The executable program can be provided with additional settings from the command line:

```
Usage: Wireworld [OPTION...]
  -w[<Filename/path>]            --world[=<Filename/path>]
  -t[<Natural number>]           --tests[=<Natural number>]
  -m[List_2D | Ordered_Lists_2D] --model[=List_2D | Ordered_Lists_2D]
  -f[<Positive number>]          --fps[=<Positive number>]
```

The individual options:

- `-w` or `--world`: Loads an alternative Wireworld, like in `./Wireworld --world="Wireworlds/XOr.bmp"` for instance. By default “`Wireworlds/Playfield.bmp`” will be loaded.
- `-t` or `--tests`: Sets the number of transition tests the program will execute on the selected wireworld before showing any graphics, like in `./Wireworld --tests=100` for instance. It will display the amount of time it took your program for this number of transitions. This will become important when you evaluate and compare your solutions. You can use `./Wireworld --tests=0` to suppress any tests. By default 25 transitions will be performed. The graphical display will still always start with the initial state.
- `-m` or `--model`: Chooses the data-structure which is to be used in this run, like in `./Wireworld --model=List_2D` for instance. By default `Ordered_Lists_2D` will be selected.
- `-f` or `--fps`: Sets the frequency at which your world and display is updated (in frames per second), like in `./Wireworld --fps=1` for instance, to slow down everything to a speed where you can see in detail what’s going on. By default 25 frames per second are set.

Of course, all of those options or any mixture of them can be given. You will need to use those options when you test both of your transition functions and measure the performance of your implementations.

Coding standards

- *No piece-wise function definitions*. While it is actually common in the Haskell community to write functions by providing multiple versions of the function which all cover only some parts of the input space (you will have noticed this praxis in our course textbook as well as in many on-line tutorials), this is considered somewhere between weird, confusing and impossible in any other programming language known to me (please educate me, if you know another language which “features” this method). The danger is that you write some parts of a function, forget others, or write them in an overlapping fashion. While the Haskell compiler actually does provide appropriate warnings in the end, it is still hard to read, and somebody who has to maintain the code will have a much harder time to see whether you covered all bases, or this patchwork of functions will make a complete function. The better alternative – if the function actually needs to branch off into different cases – is to state those cases in one spot and provide an answer to all of them right there – this way you see much easier whether the function is fully defined. Simply use case and/or guarded expressions:

~~`faculty’ :: Integer -> Integer`~~
~~`faculty’ 1 = 1`~~
~~`faculty’ 0 = 1`~~
~~`faculty’ n = n * faculty (n-1)`~~

```
faculty :: Integer -> Integer

faculty n
   | n == 0 = 1
   | n >  0 = n * faculty (n-1)
   | otherwise =
       error (“faculty undefined for “
                  ++ (show n))
```

- *Clear names for everything*. A clear name for a function combined with clear names inside the pattern which matches the inputs documents most functions well and often sufficiently:

~~`help3 :: (T, C) -> Float -> Picture`~~
~~`help3 (a, (c1, c2)) i = ..`~~

```
draw_cell ::
     (Cell, Coord) -> Float -> Picture

draw_cell (cell, (x, y)) size = ..
```

- *Usage of the Haskell Prelude* (or other standard Haskell packages) is ok (without higher order functions such as map, fold, filter, etc. – if you cannot decide whether a function is a higher order function or not, then just don’t use it), yet the same rule applies as with the modules provided here: Use only what you understood. You need to explain your design in your report and there is no room for mystery functions in there. An unexplained use of Lambda abstractions for instance (concepts outside of this course) will usually trigger a plagiarism investigation.

- *No predefined, higher order functions: If* you want to use higher order functions, then you *must* implement them yourself. This restriction is necessary to allow us to see whether you know what you are doing. Your marks come primarily from your degree of understanding – and not so much from the fact that you “got it going”.
- *Scope, or “where to put my functions”:* Where your functions appear depends on by which other functions they are used in. **Scope** is a computer science term for the context from which a function is visible and can be accessed. The usual professional reflex is to keep the **scope** small, yet to avoid replication. I.e. if a function is exclusively used by exactly one other function (and a future, wider usage is not anticipated), then consider adding it as a `where` context to this function. If you find that a function is (or might become) used by multiple other functions in different packages then place it in a separate package or a package of its own – instead of replicating it.
- *Comments where comments are necessary*. Never comment the obvious or on a line-by-line basis (assume that the reader of your code knows the language and can understand the meaning of most single code lines by just reading them). Not only would you annoy the reader with silly descriptions (like “this expression increments the index value by one”), but it also clutters your code which makes it hard to see the actual substance between all the clutter. Instead, comment in larger blocks where the functionality is no longer immediately obvious. Also document tricky sections locally, i.e. sections where the meaning or mechanism does not reveal itself easily. It is also a consideration to simply re-write those and replace them with actually easy to understand versions (instead of spending your time

documenting a weird code section). Yet sometimes there is a trade-off between efficiency and maintainability, and not everything can be re-written into a more comprehensible form.

```
~~-- This function creates a singleton world~~
~~-- It takes an element with a coordinate~~
~~-- and returns a List_2D which is the new world~~
~~ singleton_world :: Element_w_Coord e -> List_2D e~~
~~-- The function profile for singleton_world~~
~~ singleton_world element = [element]~~
~~-- singleton_world takes and elements and returns~~
~~-- a single element list with this element.~~
```

```
    Ordered_Lists_2D,
       {- the central data structure of this module:
       A list of lists where y coordinates are attached to every list at top level
       (the “lines”) and every element inside the line lists has an x coordinate
       attached to it.
       All lists are sorted in ascending order, yet the coordinates do not need to
       be consecutive (or “dense”) -}
```

Measuring your results

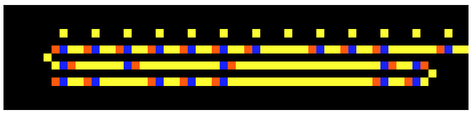
Once your transition functions work, take notes of the performance in terms of time for a certain number of transitions (use the Wireworld command line options). In order to make the measurement more stable (your machine does other things in the background and even your Haskell environment can work differently with every run) take measurements over at least 100 transitions. Do this for different size Wireworlds (you find a whole collection in the `Wireworlds` directory) as well as for both data-structures and plot your measured times against the number of cells in those worlds. What do you expect to see? Do your measurements confirm your expectations?

Program a Wireworld or extend the automaton

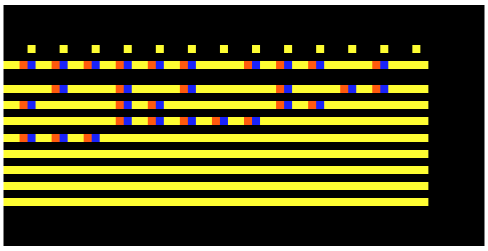
This part is mandatory for Comp1130 students and optional for Comp1100 students.

This extension part of the assignment is only to be addressed if you did all the previous, basic parts perfectly and to your fullest satisfaction. Keep in mind that a badly done assignment which addresses everything is considered worse than an assignment which addresses only the basic parts – but does it well. So if you are not yet happy with your work in this assignment so far: I suggest you rather go back and do a better job on the basics first, before you attempt this section. Otherwise: read on, there are two extension options to choose from.

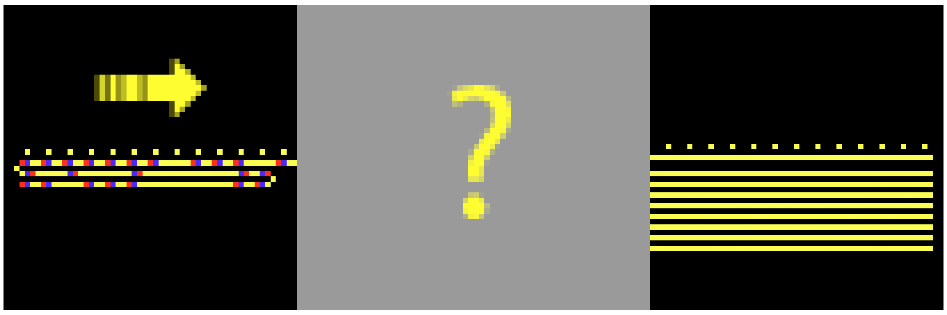
So far, you programmed the mechanics of the Wireworld automaton and watched it at work. Now you will actually program inside Wireworld. This will require a completely new look at programming. Your task is to transform this (a serial stream of bits which are “clocked” at a period of four cells):



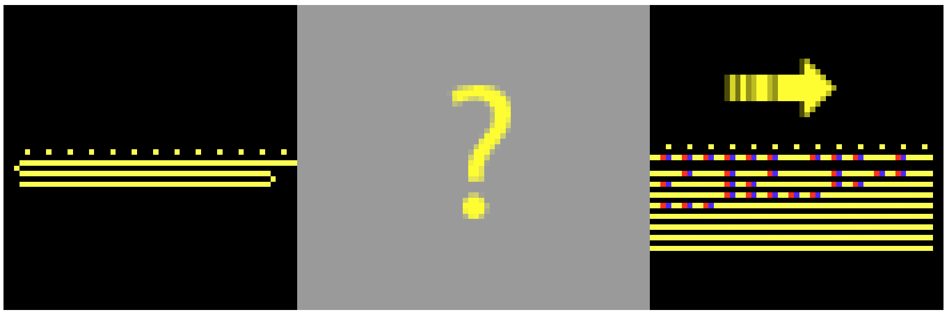
into that (which repeats the input in the top wire and represents a binary count of how many electrons have been seen so far in the eight wires below it):



Your task is to add the missing Wireworld somewhere between those two (while your output field consists of course only of wires when you design the program) and to use the smallest number of cells which you can manage. In other words, replace the grey area with the "?" in the wireworld below with an actual program (wireworld state) and connect the input field with the output field



such that it will eventually produce the required pattern rolling through the output field:



You are not restricted in the size or shape for your solution, which means that you can also connect the input and output field anywhere to your solution. The grey area above is just for illustration and does not suggest how much space you should use for your solution.

Before you come up with funny ideas: the input stream can vary (yet will always be clocked on a period of four cells), so storing the answer pattern as such is not a solution. If this was still too easy for you, try to design a counter which counts up when there is an electron and counts down when there is a gap. Careful: this is seriously hard. Either way, document your design steps and insights along the way precisely. We want to know what you learned from programming in a completely different way than what you are used to so far. Can you relate what you designed here to functional programming?

If this task is not appealing to you, then you can alternatively also deliver a new cellular automaton: Introduce one or multiple new cell types and add rules for those. Make a copy of your Wireworld directory (so that you don't alter any already working solutions for the basic part of the assignment) and name your new sources directory: `Sources_Extended`. In your new directory, you will then need to alter the packages `Data.Cell`, `Load.Pixel_To_Cell`, `Drawing.Cell_To_Colour`, `Transitions.For_Ordered_Lists_2D` (or alternatively `Transitions.For_List_2D`) to define, load, draw and transition your new cell(s). Trivial extensions like adding a second tail-cell will not count (ask your tutor if you are in doubt whether your extension might be considered trivial). The new rules do need to introduce some new quality into Wireworld. Such a new quality could for example be described as an emergent behaviour, i.e. your rule(s) should trigger patterns which are not immediately obvious. Draw a new world for your new cellular automaton to demonstrate this new quality.

Report your design concepts and findings

Write a short report which addresses the following items:

- What is your rationale for your program designs? Did you consider alternatives? Which? Why did you choose the designs which you now submit? What makes your solutions efficient and/or elegant? or: are your solutions less efficient but more readable?
- Attach your measured plots and explain why this confirms or contradicts your expectations. Would you consider changing anything in your programs to change those measurements? What specifically?
- What did you observe while programming a Wireworld (or your own new automaton) in the extension section of this assignment? Was this comparable to other programming you did so far? How was it similar or different?
- Did you learn anything (technical/scientific abilities or knowledge as well as other skills) during this assignment? What? What was the most interesting part of the assignment, and what did you spend the most time with?

You can structure your report in whichever form you see fit (and use any program you like) as long as you address the requested items and the output is of reasonably professional quality. Technically all fonts and graphics need to be "vector form", i.e. no pixelated or "bitmap" graphics (except when you attach wireworld screenshots, scans or photos – while scans or photos will need to be of reasonable high resolution). For readability, switch on kerning and ligatures in your word processor or layout program. All popular, current day programs can produce professional looking outputs (with varying degrees of effort) – one exception is "Kingsoft Office" or "WPS Office", which is known to produce output to make our eyes tear. Your plots can for instance be done on a piece of paper and scanned or photographed (as long as they are clear and complete). You final output document always needs to be in **pdf** format.

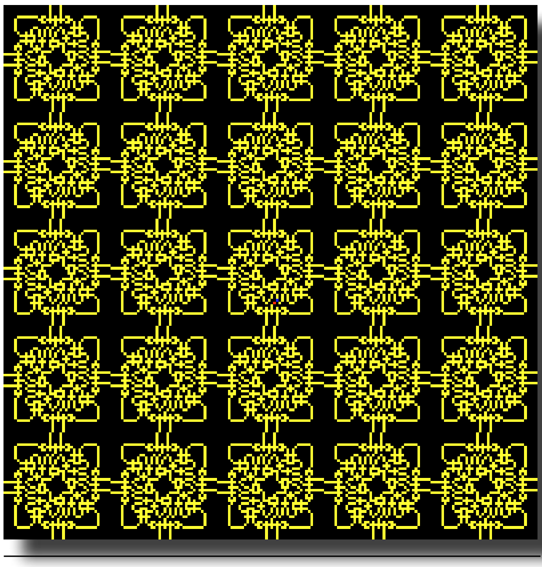
Your writing should always be concise, so avoid or remove "fluff" in your text - no marks are awarded for text length, but marks are removed for texts of a rambling nature. If you need support in technical writing in general or to manage your time at uni then check out the offers at:

https://academicskills.anu.edu.au/

If you are not sure what is expected of you: just ask us for clarification and more details.

Deliverables

Make sure your `Sources` directories have only `*.hs` files in them – delete any `*.o`, `*.hi` or executable file which might have found its way in there. Archive your `Sources` directory (and only your `Sources` directory) into a `.zip` archive, i.e. your basic submittable code archive will have the name `Sources.zip`. If you did a new cellular automaton, then also create a separate zip archive which contains a the `Sources_Extended` directory and any Wireworlds which will go with it. Call this archive `Sources_Extended.zip`. Alternatively, if you wrote a Wireworld to solve the extension task, then submit this solution as `Solved_Wireworld.bmp` file.


Langton's Ant design by Nyles Heise

Your report need to be saved as a **pdf** file. Call this `Assignment_1.pdf`. Make sure that the fonts which are used in your report are embedded.

The submission of those individual parts of your assignment (bare minimum would be `Assignment_1.pdf` and `Sources.zip`) will be done via the *SubmissionApp,* as you used in your labs before. Opening of submissions and more details about the process (if necessary) will appear on the website.

Important: the submission system allows you to re-submit as often as you want *before* the deadline. Make use of that: submit early versions for two reasons: first you know that your submits work (if you will get a success message and see your uploaded submissions) and second you are safe from last minute glitches, as you already got a version in – even if your computer explodes on the last day or your hamster is in a bad mood. Arrange your time management in order to submit one or two days before the actual deadline – submitting close to the deadline is usually a sign for a high risk strategy and bad time management and you should be concerned about this.

The deadline for the assignment will be published on the web-site.

The "late policy" for this course is: "tell us in advance or don't be late". If you see that a deadline will pose a problem for you, then please contact us with a reason for the delay, and we will work something out – otherwise: just plan to submit everything a day or two in advance and you will be fine. The submission system will automatically lock down after the deadline.

Marking

The marking is roughly divided into 60% for the code and 40% for the report. For exceptional reports or code fragments we might take the liberty to shift those percentages slightly to acknowledge extra efforts.

Marks in the code will be awarded for functionality (i.e. it works), clarity of code (i.e. we immediately understand why it works), and efficiency (i.e. it transitions at a reasonable pace).

Marks in the report will be awarded to concise writing, completeness of the report, and your understanding of your own designs and findings.

Here is a breakdown of example student cases with different marks awarded:

- *Pass:* The code compiles and works with at least one of the two data-structures. The report shows basic understanding of the problem and explains the submitted code.
- *Credit:* The code compiles without warnings and works with both data-structures. The code is well structured and well supported in the report. The measurements are fully documented and explained well.
- *Distinction:* The code is well structured and efficient. The report is excellently written and addresses all items. The measurements are close to the to be expected optimum for the individual data-structures and are well explained.
- *High Distinction:* All distinction level criteria are fulfilled and one of the extensions has been submitted and excellently documented.

Roughly you can expect to be able to reach a distinction level mark with an *excellent* submission of the basic parts alone, and a high distinction mark with an *excellent* submission including one of the extensions. Again: keep in mind that addressing an extension does not guarantee you any kind of mark – it only shifts the upper end for the potential mark.

Any plagiarism which is detected in any part of the assignment will lead to zero marks for the whole assignment and an investigation. So be strict with comments in your code and report about any collaboration and your sources. Name anybody with whom you collaborated with (and name the form and scope of the collaboration, e.g. "this function has been developed in a joint session with John von Neumann") and name any sources which you drew inspiration from for a specific part of the assignment (e.g. "this function is based on the example in section 4.3 of the course textbook"). **Never** directly exchange source code or reports as files – collaborate on paper or white boards and make sure you do not leave anything behind in the room after you leave. If your code or report (or any part of it) is found in somebody else's submission, you will become part of a plagiarism case. More details about plagiarism can be found on the course web-site.

Independent of any marking we will also look at the submissions in terms of interesting and beautiful extensions. Some of those will be shown in class, and (with your permission) potentially elsewhere.

We will also extract exceptionally elegant solutions as well as common disasters from the submitted codes and will discuss them class (with your name removed of course).

What to do if you have a problem

If you can't get it to run on your own computer ⇨ Ask your tutor for help.

If you're stuck and can't work out what to do ⇨ Take a break and do something else; staring at it is only going to frustrate you ⇨ Join a PAL session or ask your tutor for help ⇨ Take another break ⇨ Panic (and send an e-mail to the lecturer).

If you've made a bitmap world and it can't be loaded ⇨ Make sure it is a 24-bit bitmap.

In the last week of the assignment your tutor will also ask you how you are fairing and whether any last minute hints might help.

In all cases: if you have a problem which you cannot work out in reasonable time on your own, you *must* contact somebody. This can be your fellow student, the students and mentors in your PAL session, the students and tutors in your lab session, or the lecturer. All of us react to e-mail – usually much faster than your think.