1 A language for Factorio combinators

Circuit networks in Factorio are either unknown or well-known to be hard for most players, and even those who do know how to make them do what you want most often struggle to achieve the solution that is for example the most compact one (i.e. using the fewest combinators) or the most UPS efficient one.

There are multiple different reasons that lead to this problem. Creating logic circuits is akin to programming and thus similarly hard, but circuit networks in Factorio have another hurdle that prevents their widespread use: it is basically impossibly hard for a beginner to reuse someone else's circuits. Even experienced "programmers" need a lot of time to understand new designs, especially if they only come in combinator form with no accompanying explanation (I myself mostly give up on it if I'm not able to ask the creator questions about the circuit).

The reason for this lies not in insufficient methods of sharing, since blueprints are widely available and used quite often, but in the low level nature of the computation: combinators and the circuit network are akin to computer code in Assembler - the computation is there for everyone to see, but it's so incredibly hard to understand what's happening (i.e. reverse engineer it) that it simply doesn't happen in practise.

Circuit networks are usually adjusted to the specific use case, which means that even veterans may struggle to find out which "building block" was initially used to arrive at the final version. This makes it also very hard to reuse complex circuitry of others or modify it to suite your requirements. But all of this becomes much easier once access to this core information is given: just a small amount of time talking with the initial creator usually allows me to completely understand the circuit, make modifications of it, or even suggest improvements for compactness or computational efficiency - which will in turn ironically make same the job of understanding the circuit way harder for the next person to look at it.

It's thus useful to think about how to create an abstraction layer over this "bare metal" level that we're currently living in, and the idea of a programming language immediately comes to mind! But I'm certainly not the first one with this idea, so it's important to take a look at other approaches and understand why they haven't been adopted widely. There are multiple points that come to mind:

1. The language has to be compilable into blueprint form - it's no use to have a piece of text representing circuitry without being able to turn it into circuitry.

2. The above method should result in "good" results. Combinators and circuit networks are known to be Turing complete, but it's a very bad idea to compile an addition into a Turing machine and that to hundreds of combinators instead of using simply a single arithmetic combinator.

3. The programming language should actually make working with combinators easier. It's not useful to simply textify the problem without adding substantial benefit - simply adding the ability to comment on stuff is certainly not enough. The language should be able to abstract things like SR-latches, timers and pulse generators into easy to use capsules that don't have to be micromanaged.

4. The language shouldn't remove the unique abilities of combinators. Typical programming languages like say C are made to calculate with single numbers, but the circuit networks in Factorio provide us with the ability to compute in a very parallel way. The language should aim to allow people who know about combinators already to learn it fast, and help those that don't to learn about it.

2 A proposal for a language

Creating a language for circuit networks and combinators requires us to understand them first, but this isn't intended to be a tutorial on them, so we'll be very brief about them:

1. Circuit networks carry all the data in the calculation. Each network has a color (either red or green), and a value for each item and fluid in the game, as well as an extra set of values. The values are called "signal", e.g. there is an "iron-ore signal". Note that a signal is not the same as the value of the signal (i.e. the signal "iron-ore" is not the same as it's value, say -5)
2. Circuit networks lose all their values at the end of a tick after passing them to all connected entities.
3. Constant combinators simply send their configured signals to all connected networks at the beginning of each tick.
4. Arithmetic combinators save the values obtained at the end of the tick (they sum them if multiple networks provide the same signal), compute with them according to their setting, and send the result to all connected networks at the beginning of the next tick.
5. Decider combinators do the same as the arithmetic combinators, but they have a different set of transformations that get applied to the input signals.

2.1 Basic types

Note that circuit networks and combinators have the roles of variables and computations respectively. Especially note that the signals of a specific network are not the variables - they are only part of the real variable, similar to how a integer in usual programming languages consists of multiple bits. We can extract and isolate them (just like bits), but they are not the fundamental unit. Combinators on the other hand represent the fundamental operations that can be done with our variables, similar to addition and multiplication for integers.

The language will thus not have a fundamental type for signals themselves, but instead only deal with wires and combinators. Here is the complete list of fundamental circuit network types:

```
constant      decider     arithmetic
wire    pwire    connect    pconnect
```

The first three are simple the object types of combinators. The second row consists of the types of wires: `wire` is the type of a single wire (either red or green) and `pwire` (for "pair wire" or "parallel wire") represents a bundled pair of a red and a green wire each. The `connect` type represents connections between entities and wires. The `pconnect` type represents a pair of different colored connections similar to the wire case.

2.1.1 Wires

Wires alone have a few properties worth mentioning. The first being that you can combine two of the single wire types into a `pwire` type via addition:

```
pwire = wire + wire;
```

Note that the addition symbol here does not mean that the values saved in the signals on a wire get added, it's simply a combination of two wires into a bundle! We only denote them this way because the common use case is to combine two wires and feed them into a combinator - at which point their values are indeed summed.

The requirement for this combination to be valid is that both wires have different colors, which then prompts the question of how to specify a wire color. The answer is to use "`.r`" and "`g`" properties when creating a wire variable:

```
wire.r wire1;  // red wire
wire.g wire2;  // green wire
```

No circuit is really dependent on it's specific coloring, and you may later want to reuse the same component with different colorings, which is why you'll later see the color constant "`c`" which represents a list of colors to use. It's thus possible to use it to define things in a color agnostic way:

```
wire.c0 wire1;   // use the 0th color
wire.c0' wire2;  // use the opposite of the 0th color
```

The requirement for wire combination is thus simply the following (where 123 can be any constant number)

```
pwire = wire.c123 + wire.c123';  // TODO: force the output type to be pwire.c123
                                 // or allow (wire.c1 + wire.c1').c2 call
```

You can regain access to the individual wires using "`wireName.c`" and "`wireName.c'`" notation ("`.r`" and "`.g`" work for `pwire` objects made by `wire.r + wire.g`). TODO: maybe allow extraction based on a different base color (for whatever reason)? TODO: maybe remove the ability to explicitly use red and green colors, since general colored code is usually strictly better?

Wires have constrains on them that arise due to the special nature of Factorio's circuit networks. Connecting two red wire networks with another red wire destroys both individual networks and replaces them with a combined network (that usually has more complex behaviour). Note that this behaviour is akin to a change in a data structure layout in typical programming

languages, which is usually not possible during runtime - and our language therefore also doesn't allow this. Networks also lose all their values from previous ticks, which means simulating a network merely depends on the output of combinators though their connectors.

One of the many problems in understanding the behaviour of networks is that it's sometimes hard to understand where values on a network come from, which is why we'll enforce a very strict requirement on wire variables: they can only be set at most once. The strictness of this rule is also seen in the fact that it's required for wires to have unique names, which means that renaming them via assignment is not valid. The above combination and splitting of wires is also only a type level rule: the result of such operations is not allowed to be directly assigned into a new variable.

To illustrate this, here are some assignments that are strictly forbidden (given that `red` is a `wire.r`, `green` is a `wire.g` and `redgreen` is a `pwire` made by `wire.r + wire.g`), since they all simply rename networks:

```
wire.r rpart = redgreen.r;     // error: renaming of red part of a pwire!
wire.g gpart = redgreen.g;     // error: renaming of green part of a pwire!
pwire combine = red + green;   // error: renaming of a red and green wire via combination to pwire!
```

2.1.2 Combinators

As said above, combinators represent the fundamental computations that act on wires, but it's a fair bit more complex than e.g. integer operations in usual programming languages though, which is why we'll be using a more complex notation for them, too:

```
constant dummy3 = [ signal1 = 123, signal2 = -5 ];
decider dummy2 = [outputSignal = in if leftSignal > rightSignal];
arithmetic dummy1 = [outputSignal = leftSignal % rightSignal];
```

Constant combinators are simply given as a list of signals and their values (up to 18 values can be specified).
Decider combinators specify their output signals as well as the operands and the comparison operation, and indicate the output value setting by either a "`= 1`" or "`= in`" keyword. The supported operations for decider combinators are $>$, $<$, $=$, $\geq$, $\leq$ and $\neq$, where the last three can instead be written as `>=`, `<=` and `!=`.
Arithmetic combinators follow a similar syntax, with supported operators being `*`, `/`, `+`, `-`, `%`, `^`, `<<`, `>>`, `and`, `or` and `xor`.

Signal names are to be written as strings prefixed by either `i`, `f` or `v` depending on whether that signal name corresponds to an item, fluid or virtual signal. If there is no name collision it's allowed omit this prefix. The rightSignal of decider and arithmetic combinators can also be an integer instead of a signal.

Combinators can be "applied" to values on wires by writing them next to each other. You can either use the combinator name in brackets, or inline the combinator definition:

```
inputWireName1[combinatorName];
inputWireName2[outputSignal2 = 1 if leftSignal2 > rightSignal2];
```

The input wire type has to be `pwire`, but single wires types are casted automatically. This is also the place where you can use the wire addition defined in the wire section:

```
(wireName1 + wireName2)[combinatorName1];
(redName + greenName)[combinatorName2];
```

Note that the return type of the application of combinators on wires is not again a wire, but the connection type `pconnect` instead, which will be discussed in the next section. The need for the connection type arises since returning wires directly would prevent us from being able to connect multiple combinator outputs together.

Constant combinators are cast to the connection type implicitly, e.g. both of the following terms are seen as connections:

```
[ signal1 = 123, signal2 = -5 ]
[constantCombinatorName]
```

2.1.3 Connections

Connections seem similar to wires, but they behave rather differently. The most important difference between connections and wires is that we can sum connections freely:

```
wireName1[combinatorName1] + wireName2[combinatorName2] + [constantCombinatorName];
```

This sum represents the setup where multiple outputs of combinators are wired together, which sums all signal values from all outputs and puts them on a single wire.

Another difference is the restriction of connector values to a single use each, i.e. they can appear in at most one sum and are thereafter invalid. This restriction arises due to the wiring necessary in order to achieve their behaviour: summing values is only possible via wire connection, after which it's impossible to isolate the signals coming from the connector itself. Note that `pconnect` type objects underlie the same restriction, but it's possible to split them into their separate colors parts using the `[c]` property in order to use the colors individually - effectively allowing each connector to be used twice. (A nice side effect of this restriction is that the language will continue working if more wire colors are added)

Connectors automatically cast to the `wire` type (with the respective color), which makes the following assignments valid:

```
wire.r dummy2 = inputWireName1[comb1].r;
wire.g dummy3 = inputWireName2[comb2].g;
pwire dummy4 = inputWireName3[comb3];
```

Creating `pwire` (or `pconnect`) with different values for the two colors can be done by using the `.r`, `.g` properties of connections, too:

```
pwire dummy1 = inputWireName1[combinator1].r + inputWireName2[combinator2].g;
pwire dummy2 = inputWireName1[combinator1] + inputWireName2[combinator2].g;
```

Note that the first summand of the second line adds to both red and green wires of `dummy2`.

2.2 Circuits

The constructs so far are little more than a textification of combinators and offer only minimal advantages over simply designing ingame (even though it's nice to not have to worry about wire colors and be able to treat cable bundles). The next feature covers the beginning of an important part of circuit design: abstraction. Let's say we have a wire carrying multiple signals and we want to do some reoccurring computation on those, e.g. filter out only the positive ones and then set their values to a constant, say 3. The code for this would like something like this:

```
wire.r temp = input[each = 1 if each > 0].r;
pconnect result = temp[each = each * 3];
```

This code shows multiple problems with the language as it is so far, and we will fix all of them one by one. The first problem makes itself apparent after using this piece of code multiple times, but then deciding that you'd want to actually output the value 4 instead of the value 3. The slight alterations that need to be made to every copy of the code (essentially renaming input and output) make it hard to find all places where you reused this code, and even after finding all of them you'll still face the problem that altering all of them takes a lot of time and is prone to typos.

Typical programming languages solve these issues by a concept known as functions (which should be known to anyone reading this), and our language will do the same. The keyword for a function is "circuit" followed by it's return type, it's name and a list of input arguments. The result of a function is seen in the last line and denoted by the keyword "return" before the value to be returned (which has to have the specified return type). The above circuit would look like the following in code form:

```
circuit pconnect dummyName(pwire input)
{
  wire.r temp = input[each = 1 if each > 0].r;
  return temp[each = each * 3];
}
```

Note that the following piece of code is not the same function as the one above:

```
circuit pconnect dummyName(pwire input)
{
  return input[each = 1 if each > 0][each = each * 3];
}
```

It compiles correctly, since the output of the first combinator is a `pconnect`, which implicitly casts to `pwire`, which in turn is the correct input type for combinators. This code resolves the "dilemma" of having to use a dummy color, but it doesn't do it in a way we might expect it to: the compiler won't chose a color by itself, because the code tells it to use both! We instead solve this chaining problem by introducing a new syntax that explicitly tells the compiler to choose by itself:

```
circuit pconnect dummyName(pwire input)
{
  return input[each = 1 if each > 0]-[each = each * 3];
}
```

A "=" sign instead of "-" will instead explicitly tell the compiler to use both wires. Using no chaining symbol at all will compile, but issue a warning to the user to check if this behaviour is indeed intended.

This function is too short to show some other problems that may arise, so let's go on and consider the following function instead:

```
circuit pconnect dummyName(pwire input)
{
  pconnect temp = input[each = 1 if each > 0];
  return temp.r[each = each * 3].r + temp.r[each = each * 5].g;
}
```

The code does almost the same thing, with the key difference being that the result will be different on the different colors: the signal values will be 3 on the red part, and 5 on the green one. The "problem" with this function is that we may sometimes want to have a version that returns the values on different colors instead, but copy-pasting the whole function isn't ideal since the whole reason for using functions in the first place is to avoid duplication. (There would also be substantial combinatorial explosion)
We solve this issue by introducing a compile time constant argument that carries all as much color information as we want it to have:

```
circuit pconnect dummyName[c1](pwire input)
{
  pconnect temp = input[each = 1 if each > 0];
  return temp.r[each = each * 3].c0 + temp.r[each = each * 5].c0';
}
```

Where "`[c1]`" after the function name indicates that this function needs 1 color specified before being called. Calling such a function works by specifying the primary color during the call:

```
pconnect result = dummyName[r](input);
```

Let's look at two "real world" examples to understand the next language feature: given two sets of signals on different wires, we sometimes want to multiply their values pairwise (i.e. "iron-ore" from one times "iron-ore" from the other and similarly for all other signals), or we want to only let those values from the first wire, that also have a value on the second (filter) one:

```
circuit pconnect multiply30bit(pwire input)
{  // based on a * b = (a^2 + b^2) / -2 + (a + b)^2 / 2
  wire.r a2b2 = input.r[each = each ^ 2].r + input.g[each = each ^ 2].r;
  wire.r sum2 = input[each = each ^ 2].r;
  return sum2[each = each >> 1] + a2b2[each = each / -2];
}

circuit pconnect filterPositive[c1](wire.c0 signals, wire.c0' filter)
{  // assumes all signal values are positive and all filter values are 1
  wire.c0' offset = filter[each = each + 2147483647].c0';
  return (offset + signals)[each = in if each < 0]-[each = each + -2147483648];
}
```

Both of these functions will perform their task as expected, but it obfuscates a slight problem for building circuits that are sensitive to specific timings: each application of a combinator takes a update tick, which means that it's input and output don't coexist at the same time! The above filter circuit for example will calculate the "`filtered`" wire based on the "`offset`" wire of the tick before and the "`signals`" wire that existed two ticks beforehand, which is a mismatch in timings that will result in an unexpected state of the output.

Some circuits depend greatly on the specific timings of inputs and temporary variables, which is why explicitly stating those timings is supported by the language via "`:offset`" syntax on the corresponding variable and function names:

```
circuit pconnect filterPositive:3[c1](wire.c0 signals:0, wire.c0' filter:0)
{  // assumes all signal values are positive and all filter values are 1
  wire.c0' offset:1 = filter:0[each:1 = each:0 + 2147483647].c0';
  return (offset:1 + signals:0)[each:2 = in if each:1 < 0]-[each:3 = each:2 + -2147483648];
} // TODO: maybe number combinators differently?
```

The numbering follows the simply rule that each combinator application increases the maximal input number by one, and each function application increases it by the number specified at it's name. This will be enforced by the compiler, but it still means that it's the job of the programmer to make sure that the timings match where they should match. (Enforcing timings is a bad idea, since you sometimes explicitly want to match stuff at "wrong" timings, e.g. when making a throughput counter.) This also means that you can freely cast between numbered and unnumbered variables.

An immediate generalization of timing numbers is given by manually increasing the timing number when using the variable to indicate a deliberate delay. The compiler would thus only check that each use of variables carries a number bigger or equal to the one that was used during the creation of the variable. The above filter function could thus "correct" it's timings by simply increasing a single number by 1: TODO: does this QoL feature cause any problems elsewhere?

```
circuit pconnect filterPositive:3[c1](wire.c0 signals:0, wire.c0' filter:0)
{  // assumes all signal values are positive and all filter values are 1
  wire.c0' offset:1 = filter:0[each:1 = each:0 + 2147483647].c0';
  return (offset:1 + signals:1)[each:2 = in if each:1 < 0]-[each:3 = each:2 + -2147483648];
}
```

TODO: can we maybe enforce stricter type checking rules for timings to make them more useful (by being less error prone) without restricting their usability?

2.3 Networks

The circuits presented so far act in a very functional way: they assume that the input just has to be transformed into the output during computation, but it only cares about the eventual result. The reality is that each wire and connector carries values during each tick of the simulation, which means that we're able to do do something like the following:

```
circuit connect.c0 memory:1[c1](wire.c0 toAccumulate:0)
{
  pwire memory:1 = (memory:0.c0' + toAccumulate:0)[each:1 = in if any != 0];
  return memory:1.c0;
}
```

This circuit function fulfils all the conditions we set so far and would thus normally compile just fine. But there is a glaring issue: it's unclear what `memory:0` is supposed to be, since we only define `memory:1` (and thus all higher timings of memory via delay).

Many circuits can avoid such weird delays, but there is one class of circuits that will never be able to: it's the class of circuits that have a circular wiring topology (i.e. the input of some combinator is directly or indirectly dependent on itself). It would be very unwise to dismiss such circuits, since the simplest case of those (a combinator whose in- and output are connected to each other directly) constitutes a memory cell. Memory cells are important because they are the only way to not lose all network content after a single tick, and they are (to my knowledge) impossible to create without a circular network topology.

We could simply allow these circuits with the above syntax, but it won't solve the problem of what should happen during the first call of the function: the "previous" value `memory:0` would be undefined, and it's thus impossible to predict reliably what will happen (at least not in the general case). The default state of wires is the one where they do not carry any signals, but it's not desirable to restrict our language to only allow zero valued wires as an initial condition (one of my throughput measurers assumes that a certain memory cell contains specific values at the very beginning).

The solution is to remove the definition of the corresponding variable from the function itself, and make it a global variable itself, which then allows us to specify initial conditions there (by using the constant combinator syntax, which casts to `pconnect` and then in turn to `pwire`):

```
pwire memory = [ ];
circuit connect.c0 memory:1[c1](wire.c0 toAccumulate:0)
{
  memory:1 = (memory:0.c0' + toAccumulate:0)[each:1 = in if any != 0];
  return memory:1.c0;
}
```

This solves our problem of initial conditions, but it introduces another one: the `memory` variable is now global, which means that the circuit may only be used a single time in the whole program, since multiple uses would erroneously try to write to the same network twice!

We could solve this issue by moving the definition back into the circuit function and marking it with a new keyword (the `static` keyword from C♯ comes to mind), but we use this method because it motivates us to introduce another key abstraction feature used in nearly every programming language: custom data structures. It's tedious to manage all your data handling wires separately, and I attribute me never seeing multiple cable data use to this fact (say a fractional calculator, where one cable holds the numerators and another the denominators).

Our language will allow to do this in by using the "`network`" keyword, which defines a new type which allows you to capsule data similar to classes and structs in other languages:

```
network memory[c1]
{
  wire.c0' data = [ ];
  circuit connect.c0 add:1(wire.c0 toAccumulate:0)
  {
    data:1 = (data:0.c0' + toAccumulate:0)[each:1 = in if any != 0];
    return data:1.c0;
  }
}

memory newCell = memory[r];
newCell.add([ 'iron-ore' = 5 ].r);
```

You can add member functions to your networks to allow for easy management of them, or you can keep them strictly as data containers and simply pass them to functions. Note that all the wires inside a network are still bound to their restriction of needing to be set once per update cycle, so I'd suggest to design your functions on them in a way that makes it clear which combinations of functions can be called during the same tick.

Network accept color constant parameters similar to functions. These are accessible everywhere inside the network scope, including in the member functions themselves. Member functions are also allowed to ask for additional color parameters by

using a special syntax, e.g. "`[c+3]`" to ask for three more colors. Addressing those is done by e.g "`.c+1`" to make it easy to change the amount of either without influencing the calls to the others. TODO: should the range be "`.c+0`" to "`.c+(n-1)`" or "`.c+1`" to "`.c+n`"?

2.4 Meta programs

The language so far contains everything that is necessary to create almost any circuit without being more complex than necessary. An exception to this is found when trying to create say a timer circuit:

```
network timer[c1]
{
  wire.c0' time = [ ];
  circuit connect.c0 update()
  {
    pconnect next = (time + [ 'T' = 1 ].c0)['T' = in if 'T' < 100];
    time = next.c0';
    return next.c0;
  }
}
```

This timer uses a hardcoded signal and timing length even though the circuit doesn't depend on it at all. It's not hard to modify the language to support those, since colors are already constants that are passed around, but there is one reason why we'll delay adding number and signal constants for now: color constants are very simple in their use, while numbers and signals would immediately ask for methods to calculate thing with them, e.g. calculate the constants set into specific combinators based on the constant integer passed into the method.

The ideal language would allow for a complete meta programming language that allows you to create networks and circuits synthetically rather then having to type all of them. This would add an incredible amount of complexity to the language and make it harder to create the language, which is why we're going to postpone these features to a version 2.0.