I've just released a new version of AirScript (v0.7) and it is also a part of the v0.7.4 of genSTARK library. In this release, AirScript gains two major new features:

1. Ability to split execution trace into subdomains and run multiple parallel loops in these subdomain.
2. Ability to import functions from AirAssembly modules. So, this is the first release that supports script composition - something that I've been working toward over the last few months.

Here is an example of AirScript that defines a computation for verification of Merkle proofs (finally, this example is compact enough that I can use it instead of MiMC example

)

```
import { Poseidon as Hash } from './assembly/poseidon128.aa';

define MerkleBranch over prime field (2^128 - 9 * 2^32 + 1) {

secret input leaf       : element[1];      // leaf of the merkle branch
secret input node       : element[1][1];   // nodes in the merkle branch
public input indexBit   : boolean[1][1];   // binary representation of leaf position

transition 6 registers {
   for each (leaf, node, indexBit) {

      // initialize the execution trace for hashing (leaf, node) in registers [0..2]
      // and hashing (node, leaf) in registers [3..5]
      init {
         s1 <- [leaf, node, 0];
         s2 <- [node, leaf, 0];
         yield [...s1, ...s2];
      }

      for each (node, indexBit) {

         // based on node's index, figure out whether hash(p, v) or hash(v, p)
         // should advance to the next iteration of the loop
         h <- indexBit ? $r3 : $r0;

         // compute hash(p, v) and hash(v, p) in parallel
         with $r[0..2] yield Hash(h, node);
         with $r[3..5] yield Hash(node, h);
      }
   }
}

enforce 6 constraints {
   for all steps {
      enforce transition($r) = $n;
   }
}

}
```

(for those interested, the referenced poseidon128.aa

AirAssembly file is here).

This is a fully functional example, and you can find the code that uses the above script to generate and verify STARK proofs here.

The way it works in the background is that the compiler combines AirScript source code with imported AirAssembly functions and generates a single AirAssembly module for the computation. genSTARK then uses this AirAssembly module to generate/verify proofs.

There as still some limits to how the composition works. For example, right now, you can make function calls only from the inner-most loops. This limitation will be removed in the future. But even with this limitation, writing moderately complex STARKs is way simpler than before.

So, my future plans are:

1. Start building a standard library of AirAssembly components so that people don't have to write AirScript for such things as hash functions, Merkle branch verifications, Schnorr signatures etc.
2. Improve and make the composition features more powerful.
3. Optimize AirAssembly output generated by the compiler (right now, it is quite poorly optimized).

As always, any feedback or help is welcome!