University of Sussex
Informatics Spring 2020

Limits of Computation
Exercises 4

`WHILE`-programs, `WHILE`-decidability, `WHILE`-computablity
(Lectures 7–9)

1. By writing a `WHILE`-program, show that $A = \{ 4, 6, 8 \} \subseteq \mathbb{D}$ is `WHILE`-decidable. Test your program by running it in `hwhile`.

2. Show that *any* finite set $A \subseteq \mathbb{D}$ is `WHILE`-decidable.
Hint: assume without loss of generality that the finite set has n elements $A = \{ d_1, d_2, \ldots, d_n \}$ and write the decision procedure for this A.

3. Show that, if $A \subseteq \mathbb{D}$ is `WHILE`-decidable then so is the complement of A, that is $\mathbb{D} \setminus A$.
Hint: assume A is `WHILE`-decidable and thus we have a `WHILE`-program `p` that decides A. Now write a `WHILE`-program `q` that decides the complement of A. Of course, you can and should use `p`.

4. Why is any `WHILE`-decidable set automatically `WHILE`-semi-decidable.

5. Write a `WHILE`-program `equal` that does not use the built-in equality (but can use all other extensions). The program `equal` takes a list of two trees `[l,r]` and tests whether the trees are equal, i.e. whether `l` = `r`. The function `equal` can be defined recursively as follows:

$$\texttt{equal}([\text{nil}, \text{nil}]) = \texttt{true}$$

$$\texttt{equal}([\text{nil}, \langle\, \texttt{l.r} \,\rangle]) = \texttt{false}$$

$$\texttt{equal}([\langle\, \texttt{l.r} \,\rangle, \text{nil}]) = \texttt{false}$$

$$\texttt{equal}([\langle\, \texttt{l.r} \,\rangle, \langle\, \texttt{s.t} \,\rangle]) = \texttt{equal}([\texttt{l}, \texttt{s}]) \wedge \texttt{equal}([\texttt{r}, \texttt{t}])$$

Unfortunately `WHILE` does not provide any recursive features. So your implementation has to traverse both input trees using a while-loop. One way to do this is to generalise the equality test to stacks of pairs of trees represented as a list of pairs of trees:

$$\begin{aligned}
\texttt{equalG}([]) &= \texttt{true} \\
\texttt{equalG}([[\text{nil}, \text{nil}], \texttt{S}]) &= \texttt{equalG}(\texttt{S}) \\
\texttt{equalG}([[\text{nil}, \langle\, \texttt{l.r} \,\rangle], \texttt{S}]) &= \texttt{false} \\
\texttt{equalG}([[\langle\, \texttt{l.r} \,\rangle, \text{nil}], \texttt{S}]) &= \texttt{false} \\
\texttt{equalG}([[\langle\, \texttt{l.r} \,\rangle, \langle\, \texttt{s.t} \,\rangle], \texttt{S}]) &= \texttt{equalG}([[\texttt{l}, \texttt{s}], [\texttt{r}, \texttt{t}], \texttt{S}])
\end{aligned}$$

(If the input list contains more than two trees, those following the first two shall be simply ignored.) One can now define

$$\texttt{equal}(\texttt{L}) = \texttt{equalG}([\texttt{L}])$$

The definition of `equalG` is a so-called *tail-recursive* definition which means it can relatively straightforwardly be transformed into a while loop like so

```
equalG read L {
 res:= true;
 while L {
  X:= hd L;
  ...
  res:= ...
  if res
   { L := tl L  // tail recursive call
   }
   else
   { res:= false;
     L:= nil   // loop exit
     }
  }
}
write res
```

where some bits (represented by ...) have been left out for you to fill in. Test your program in `hwhile`.