University of Sussex
Informatics

Spring 2021

Limits of Computation
Exercises 4

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

1. By writing a `WHILE`-program, show that $A = \{\ulcorner 4 \urcorner, \ulcorner 6 \urcorner, \ulcorner 8 \urcorner\} \subseteq \mathbb{D}$ 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. You can't run it, of course, until you know what the elements $d_1, d_2, \ldots$ are.

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 $\mathtt{l} = \mathtt{r}$. The function `equal` can be defined recursively as follows:

$$\mathtt{equal}([\mathrm{nil}, \mathrm{nil}]) = \mathtt{true}$$

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

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

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

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:

$$\texttt{equalG}([]) = \texttt{true}$$

$$\texttt{equalG}([[\text{nil}, \text{nil}], \texttt{S}]) = \texttt{equalG}(\texttt{S})$$

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

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

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

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

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

The definition of `equalG` is a so-called *tail-recursive* definition which means that the recursive call is at the top level, which in turn means, it can relatively straightforwardly be transformed into a while loop like so:


```
equalG read L {
 res:= true
 while L {
  X:= hd L;
  s := hd X;
  t := hd tl X;
  if s {
    if t {
    ...
        }
      else {
      ...
   }
  }
}
write res
```

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