University of Sussex
Informatics

Spring 2025

Limits of Computation
Assignment 1 (Deadline 6.03.2025, 4pm)

You need to submit this coursework electronically at the correct E-submission point. You must submit a zipped directory that contains three files with the exact names as described below:

- a *pdf* file `questions.pdf` containing the answers to Questions 1–4. Please make sure that *all* answers for Questions 1–4 are in *that single document*. In case you use Word, please ensure that you convert your file into a pdf document before submission[1].
- a runnable `WHILE` source file `concat.while` that contains the answer to Question 5a.
- a runnable `WHILE` source file `STEPn.while` that contains the answers to Question 5b.
- Any other files in your directory will be ignored, so include no other programs or files. Note that both `WHILE`-programs, for Question 4 and 5, are only allowed to call macros that are published on our Canvas site (or your answer for Question 5a) and both must run in `hwhile` or `whilers`. For marking your assignment it is essential that you follow the above rules.
- Please use a standard zip program to zip the directory. If you work on a Unix machine or a Mac use the (normally) pre-installed zip program. If you use a Windows machine, use WinZip or 7-Zip. Please do not use other compression programs or formats as this might give you 0 marks as the markers may be unable to unzip.
- Please do not write your name anywhere, but it is advisable to *include your candidate number as comment* in each submitted file.
- Please make sure you *check after submission* that you actually have submitted the correct zipped directory of files a the correct submission point.

[1] In Word, this works usually by using the printing menu and then saving as pdf instead of sending to a printer.

YOU MUST WORK ON THE ASSIGNMENT ON YOUR OWN! The standard rules for collusion and plagiarism apply and any cases discovered will be reported and investigated.

Questions

1. A *list of lists of numbers* is a list such that all its elements are lists of numbers. Consider the following examples:

   - `[[3,5],[],[1,1,1]]` is a list of lists of numbers;
   - `[[[3,5]]],[],[1]]` is not a list of lists of numbers due to the first element;
   - `[2,[1]]` is not a list of lists of numbers due to the first element not begin a list of numbers, but note that the tree that encodes this list also encodes `[[0,0],[1]]` which *is* a list of list of numbers.

   Note that an empty list can always be considered a list of numbers and a list of lists of numbers.

   The (semantics of) language `WHILE` uses the binary tree data type $\mathbb{D}$. Consider the following elements in $\mathbb{D}$:

   (a) $\langle\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle\mathrm{nil}.\mathrm{nil}\rangle\rangle.\langle\mathrm{nil}.\mathrm{nil}\rangle\rangle\rangle$

   (b) $\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\mathrm{nil}\rangle.\langle\mathrm{nil}.\mathrm{nil}\rangle\rangle\rangle\rangle\rangle\rangle$

   (c) $\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\mathrm{nil}.\mathrm{nil}\rangle.\langle\langle\mathrm{nil}.\langle\mathrm{nil}.\mathrm{nil}\rangle\rangle.\langle\mathrm{nil}.\mathrm{nil}\rangle\rangle\rangle\rangle\rangle$

   Answer for EACH of the given trees (a)–(c) above BOTH of the following questions:

   i Does it encode a *list of numbers*? If it does which is the corresponding list of numbers?

   ii Does it encode a *list of lists of numbers*? If it does which is the corresponding list of lists of numbers?

   Use our encoding of datatypes in $\mathbb{D}$ as explained in the lectures. [18 marks]

2. Given the following `WHILE`-program as data d, write a source `WHILE`-program p such that $\ulcorner p \urcorner = d$.

```
[0, [
    [:=, 1, [quote, nil]],
    [while, [var, 0], [
        [:=, 1, [hd, [var, 0]]],
        [if, [var, 1], [
            [:=, 2, [cons, [var, 1], [var, 2]]]
        ], []],
        [:=, 0, [tl, [var, 0]]]
     ]],
    [:=, 1, [cons, [quote, nil], [var, 1]]]
], 1]
```

Make sure your program has correct core WHILE-syntax. [14 marks]

WHILEfor – Extending WHILE with an Iterator

For the following questions, we enrich our WHILE-language with a list-iterator (such containers you will know from Java where there are iterators for collection types). The extended language shall be called WHILEfor.

The syntax of our list iterator statements is defined below adding a case to the command case of the grammar for the WHILE-language from Lecture 3, Slide 17.

$\langle command \rangle$ ::= ... | /* WHILE-commands */
foreach $\langle var \rangle$ **in** $\langle expression \rangle \langle block \rangle$ /* list iterator */

The semantics of the iterator command `foreach X in E B` is as follows: first evaluate E to a list (recall that this is always possible). For each element d of this list execute block B assuming that variable X (occurring in B) has value d. The iteration works from the front of the list towards the end of the list. If the list is empty the block B is never executed and the iterator does nothing. Note that E is evaluated only once at the beginning, it is never re-evaluated, even if it contains any variable that is changed by B! After execution, the variable X contains the last element of the evaluated list E unless it is assigned a value in B. In the latter case it will retain the last value assigned to it.

Let us define store $\sigma = \{\mathtt{X}{:}\mathrm{nil}, \mathtt{Y}{:}\ulcorner[0,1,2]\urcorner, \mathtt{Z}{:}\mathrm{nil}\}$. Here are some examples of the semantics of the iterator:

`foreach X in Y {Z:= cons X Z}` $\vdash \sigma \to \{\mathtt{X}{:}\ulcorner 2\urcorner, \mathtt{Y} : \ulcorner[0,1,2]\urcorner, \mathtt{Z} : \ulcorner[2,1,0]\urcorner\}$

`foreach X in hd Z {Z:= cons X Z}` $\vdash \sigma \to \sigma$

`foreach X in Y {Z:= cons nil Z}` $\vdash \sigma \to \{\mathtt{X}{:}\ulcorner 2\urcorner, \mathtt{Y} : \ulcorner[0,1,2]\urcorner, \mathtt{Z} : \ulcorner 3\urcorner\}$

`foreach X in Y {X:= cons nil X; Z:= cons X Z}` $\vdash \sigma \to$
$\{\mathtt{X}{:}\ulcorner 3\urcorner, \mathtt{Y}{:}\ulcorner[0,1,2]\urcorner, \mathtt{Z} : \ulcorner[3,2,1]\urcorner\}$

`foreach X in Y {Z:= cons X Z; Y:= tl Y}` $\vdash \sigma \to \{\mathtt{X}{:}\ulcorner 2\urcorner, \mathtt{Y}{:}\mathrm{nil}, \mathtt{Z}{:}\ulcorner[2,1,0]\urcorner\}$

$$\texttt{foreach X in Y \{X:= hd tl Y; Z:= cons X Z\}} \vdash \sigma \rightarrow \{\texttt{X}:\ulcorner 1 \urcorner,\ \texttt{Y}:\ulcorner [0,1,2] \urcorner,\ \texttt{Z}:\ulcorner [1,1,1] \urcorner\}$$

We also extend "programs-as-data" to include this form of iterator. The corresponding rule for the encoding (see the encodings given in Lecture 6 on Slide 18) looks as follows

$$\ulcorner \texttt{foreach X in E B} \urcorner = [\texttt{for}, \mathit{varnum}_{\texttt{X}}, \ulcorner \texttt{E} \urcorner, \ulcorner \texttt{B} \urcorner]$$

where `for` is a new atom. In order to be able to deal with this in `hwhile` or `whilers` (our `WHILE` interpreters) that do not recognise `@for`, let us fix the encoding of `for` to be number `4`. So in program as data representation of `WHILEfor`-programs you *must* use `4` to represent the foreach construct, e.g.

```
foreach X in Y {Z:= cons X Z}
```

is represented as `[4,0,[var,1],[[:=,2,[cons,[var,0],[var,2]]]]]`.

3. Let `prog` be the `WHILEfor`-program in Figure 1.

```
prog read L {
 foreach X in L
 {
  while X {
    Y:= cons nil Y;
    X:= tl X
    }
 }
}
write Y
```

Figure 1: `WHILE`-program `prog`

   Explain what program `prog` in Figure 1 returns as output for a given input $d \in \mathbb{D}$. In other words, state what $[\![\texttt{prog}]\!]^{\texttt{WHILEfor}}(d)$ is for any $d \in \mathbb{D}$.

   Do not narrate how the program executes step by step, but provide/describe the *function* it implements. Your description should explain the result for *any* input $d \in \mathbb{D}$, and thus should refer to the input tree. [20 marks]

4. Can we decide more problems with `WHILEfor` programs than we can decide with `WHILE`-programs? Explain your answer carefully *without* referring to Church's Thesis. [14 marks]

5. We would like to extend the self-interpreter `u.while` for `WHILE` (discussed in Lecture 7 and available from our Canvas site) so that it can also interpret `WHILEfor` programs in abstract syntax. So in order to do this please answer the following two parts:

   (a) You will find a concatenation program useful later. So write a `WHILE`-program `concat` that views its input as a list of lists and concatenates those lists in the order given. So, consider for instance, if we were running `concat` on input `[[0,1,2],[3,4,5],[6,7,8]]` this would return the result `[0,1,2,3,4,5,6,7,8]`.

   Empty lists in the input list shall be ignored, so e.g. running `concat` on input `[[0,1,2],[],[6,7,8]]` shall return the following result: `[0,1,2,6,7,8]`. Note that the single argument list may contain an arbitrary number of lists, not just 3. For instance, running `concat` on input `[[[1],2]]` shall return `[[1],2]`, whereas running `concat` on input `[[1],2]` shall return the result `[1,0,0]`.

   Submit the answer to this question in a separate file `concat.while` that is runnable in `hwhile` or `whilers`. You may use any language extension features available in those interpreters. In your `concat` program you may call any program *published on the Canvas site*but no other program. Test your answer program using `hwhile` or `whilers` before you submit. Programs that don't run will get zero or very few marks. Add comments to your code where appropriate to help the marker understand your thought process. [14 marks]

   (b) In order to extend the self-interpreter `u.while` so that it is able to interpret the language `WHILEfor`, you *only need to edit* the implementation of the step macro `STEPn.while` (which will be released on our Canvas site just after the last seminar at the end of Week 4). Note that you have to *add* code (at the right places) for the required additional iterator, the existing code does not need to be changed. The step macro must still work fine for `WHILE`-programs that don't use the iterator. Don't forget to use the number 4 to encode the atom `@for`, and if you need a new auxiliary atom make sure it won't clash with any of the atoms used already in the step macro (Tip: avoid prime numbers).

   Submit the answer to this question in a separate file `STEPfor.while` that is runnable in `hwhile` or `whilers`. You may use any language extension features available in those interpreters. In your answer program you may use macro calls to the program from Question 5a above

and any program *published on the Canvas site*. In the latter case, please don't include them in your submission. You must not call any other programs. Test your programs using `hwhile` or `whilers` before you submit. Programs that don't run will get zero or very few marks. Add comments to your code where appropriate to help the marker understand your thought process.

One must be able to successfully run the universal program `u` as interpreter for `WHILEfor` changing its `STEPn` macro call to a `STEPfor` macro call. This is how we will test your answer and you should test yours in this way before submitting. Also make sure you test that your code does not lead to non-termination. This often happens when one does not clear the command stack properly. [20 marks]