Programming Language Design
Mandatory Assignment 4 (of 4)
Version 1.1

Torben Mogensen, Hans Hüttel, Fritz Henglein

March 22, 2021

This assignment is *individual*, so you are not allowed to discuss it with other students. All questions should be addressed to teachers and TAs. If you use material from the Internet or books, cite the sources. Plagiarism *will* be reported.

Assignment 4 counts 40% of the grade for the course, but you are required to get at least 33% of the possible score for every assignment, so you cannot count on passing by the later assignments only. You are expected to use around 35 hours in total on this assignment.

The deadline for the assignment is **Friday April 9 at 16:00 (4:00 PM)**.

The assignments consists of several exercises, specified in the text below. You should hand in a single PDF file with your answers and a zip-file with your code. Hand-in is through Absalon. The assignment answers must be written in English.

Fixed in v.1.1: query in A4.6

A4.1) (20%) Limits

Consider an imperative language SIS (Simple Imperative Statements) with the following grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{Var++} \\
S & \rightarrow & \textbf{Var--} \\
S & \rightarrow & S\texttt{;}\ S \\
S & \rightarrow & \texttt{for}\ \textbf{Var=Var}\ \texttt{to}\ \textbf{Var}\ \{\ S\ \}
\end{array}
$$

where **Var** denotes a variable name, which is a nonempty, finite sequence of alphabetic letters.

The semantics are as follows:

- A variable can hold a 32-bit unsigned integer.
- The variables `a`, `b` and `c` are the only global variables and initially hold the input to the program.
- At the end of execution, `a`, `b` and `c` hold the output to the program.
- x`++` increments the variable x. $2^{32}-1$ is incremented to 0 (so arithmetic is modulo 2^{32}).
- x`--` decrements the variable x. 0 is decremented to$2^{32}-1$.
- s_1`;` s_2 executes s_1 and then s_2.
- `for` i`=`x `to` y `{ s }` creates a new variable i that has scope inside the body s. i is initialised to the value of x. The body s is executed 0 or more times until i is exactly equal to y, at which point the loop terminates and i is no longer available. The following restriction applies: After any execution of s, i is not allowed to be equal to x. This is checked at runtime, and if i becomes equal to x, a run-time error is issued.

An example of a program in SIS is the following, which starts with $a=n$, $b=0$, $c=0$ for some n and ends with $a=n$, $b=0$ and c equal to $2n$ modulo 2^{32}.

```
for  i=a  to  b  {  c++;  c++;  i——  };
```

a. Write a SIS program that starts with$a=n$, $b=0$, $c=0$ and ends with $a=n$, $b=0$ and c equal to the sum of all numbers from 1 to n (modulo 2^{32}).

b. Show that SIS is a reversible language. More precisely, given any program p is there always a program p', such that when p starts with $a = n_1$, $b = n_2$, $c = n_3$ and terminates without error yielding $a = n'_1$, $b = n'_2$, $c = n'_3$ then starting p' with $a = n'_1$, $b = n'_2$, $c = n'_3$, it also terminates without error and yields $a = n_1$, $b = n_2$, $c = n_3$.

   Argue your answer by showing how a program p can be inverted to a program p' such that the above holds. Explain for the loop construct why (or why not) the restriction is required for reversibility.

c. Is the problem of whether or not a SIS program terminates (with or without error) in finite time, regardless of input, a trivial, decidable, or undecidable property? Give convincing arguments why this is so. Hint: Your argument needs to touch both reversibility, the range of numbers in variables, and the restriction on the loop.

A4.2) (15%) Consider break-statements, as used in C. A break jumps to right after the closest enclosing loop or multi-way branch. (PLDI, Section 7.2.4). Consider specifically the program

```
i = 10; j = 4; n = 0;          // Init
while (i > j) {                // Test
  if (i % j == 0) break;       // CBreak
  i--;                         // IDecr
  n++;}                        // NIncr
  printf("%d", n);             // Print
```

where the while-loop is exited in line `CBreak` if `j` divides `i`.

(a) Write the sequence of lines `Init, Test, ...` executed in the program above. What is the value of `n` printed?

(b) Eliminate the `break` statement in the while-loop above without using a `goto`- or `continue`-statement. The resulting while loop must retain both semantics and syntactic similarity to the original while-loop.

(c) Describe a general method for eliminating break-statements inside loops without introducing goto-statements. It is sufficient to describe it for loop bodies that are either assignments or they are if-then statements, if-then-else statements or while-loops, where the statements inside of these also can have any of these forms. The semantics must be the same; neither syntactic similarity nor efficiency of the resulting code are required.

A4.3) (30%) Embedding Troll in PLD LISP

You are to implement a subset of Troll (see `https://topps.diku.dk/~torbenm/troll.msp` and `http://hjemmesider.diku.dk/~torbenm/Troll/manual.pdf`) as an embedded DSL i PLD LISP. Note that a new version has been uploaded to Absalon. The only difference is that a larger set of characters are allowed in symbols.

Only the random roll semantics should be implemented. You can use the simple linear congruential random number generator in the file `prng.le` that you can find in the new `LISP.zip` file on Absalon.

You should implement the following operations from Troll:

n
x
`d` e
e_1 `d` e_2
e_1 `#` e_2
e_1 `+` e_2
e_1 `-` e_2
e_1 `<` e_2
e_1 `>` e_2
`sum` e
`count` e
$\{e_1, \ldots, e_n\}$
`min` e
`max` e
`least` $e_1\ e_2$
`largest` $e_1\ e_2$
`choose` e
$x := e_1$ `;` e_2
`accumulate` $x := e_1$ `while` e_2

where n is a number, x is a variable, and e_i are Troll expressions.

a. Describe how the syntax of each of the above is represented as embedded syntax in PLD LISP. Discuss nontrivial choices.

b. Implement the random roll semantics for the above subset of Troll. You can assume that Troll programs are well formed. You should be able to specify how many random results you want. Discuss nontrivial aspects of the implementation. Your implementation should be included as a file `Troll.le` with your report.

c. Write a short guide of how to use your implementation to generate random results. Show examples of use. This should test all constructions and at least include equivalents of the following Troll expressions:

- `d12+d8`
- `min 2d20`
- `max 2d20`
- `sum largest 3 4d6`
- `count 7 < 10d10`
- `choose {1,3,5}`
- `accumulate x:=d6 while x>2`
- `x := 3d6; (min x)+(max x)`

Generate 10 different random results for each of these and show the PLD session in your report.

d. If you did not manage to get everything to work, describe what is missing and what thoughts you have about implementing the missing parts.

A4.4) (10%) Consider a programming language with interval types over integers.

Examples of interval types would be

```
type DanishGrades = −3..12
type OldDanishGrades = 0..13
```

(a) Give a precise definition of a subtype ordering $\sqsubseteq$ on interval types and prove that it is a partial order.

(b) We can define a type constructor Association on an interval types and some other type. Values of type Association (T1,T2) are lists of pairs whose first elements are values of type T1 and whose second elements are values of type T2 and such that all first elements are distinct.

As an example, the list

```
[ (−3,'f'), (0,'f'),(2,'e'),(4,'d'),(7,'c'),(10,'b'),(12,'a') ]
```

is a value whose type is Association(DanishGrades,Char).

We would now like to extend the subtype ordering to types of the form Association(T1,T2) where T1 and T2 are interval types, where we assume the subtype ordering $\sqsubseteq$ on interval types that you defined in the solution to the subproblem above.

The type constructor Association takes two types as arguments. Should the constructor be covariant or contravariant wrt. the subtype ordering $\sqsubseteq$ for its first argument? Should it be covariant or contravariant wrt. the subtype ordering $\sqsubseteq$ for its second argument? You must justify your answers.

A4.5) (15%) We extend our tiny imperative language of Section 12.6.2 of PLDI with a bounded loop construct such that the formation rules for statements are now

$$s ::= \cdots \mid \texttt{for}\ x := n_1\ \texttt{to}\ n_2\ \texttt{do}\ s_1$$

The information description of this construct is that `for` $x := n_1$ `to` n_2 `do` s_1 the variable x is set to n_1 and the loop body s is executed, if $n_2 > n_1$. Otherwise, the loop terminates. If the body is executed, the value of x is incremented by 1 and we enter the loop again.

Extend the big-step semantics for the tiny imperative language with transition rules that describe this.

A4.6) (10%) Here is a Prolog program.

```
friend(a,b).
friend(b,c).
friend(c,d).

knows(X,Y) :- knows(X,Z),knows(Z,Y).
knows(X,Y) :- friend(X,Y).
```

(a) The query knows(a, d) looks innocent, but it fails to terminate. Explain by means of the resolution algorithm why this happens.

(b) Modify the program such that the query knows(a, d) will terminate properly and explain why your modification works.