CPSC 311: **Practice** Midterm Exam #1

CWL ID for GradeScope: _____@students.cs.ubc.ca

- Instructions true for both the practice exam and actual exam:
  - **ALL PROBLEMS** refer to the `A3L` language and parser/interpreter attached. (This is identical to the code handout for the tutorial except with less critical sections of the code—including most tests and error handling—removed for space.)
  - **EACH PROBLEM** explores a change to the given language. The change discussed in a particular problem should be considered for **that problem alone**. Each problem starts "fresh" from the provided language and parser/interpreter.

EXAM NOTE: *The problem titles match between actual exam questions and related practice exam questions. We include brief **EXAM NOTES** to remind you how the exam problems correspond with practice problems.*

EXAM NOTE: *There are 30 marks available in this exam. A pace of about 1 minute per mark will give you ample time to recheck your questions.*

Instructions for the practice exam alone:

We anticipate asking questions similar in structure or spirit on the actual exam. Students who have carefully worked through and understood these problems will be at an **immense** advantage in the exam over those who have not. In each question, we include at least one **PRACTICE NOTE** to describe how the actual exam may relate to this practice exam. Note also that while the problem titles are not meaningful, they will match between actual exam questions and related practice exam questions. (On the actual exam, we include brief **EXAM NOTES** to remind you how the exam problems correspond with practice problems.)

Note that we may not ask all of these questions, but at this time this practice exam does explore the set of questions under consideration. Whe intend to ask an extra question that we will not be discussing in this practice midterm, but we do intend to follow the same distribution of marks per question.

We did *not* scale the practice exam to be of an appropriate length for 50 minutes.

1 Determining the scope of inquiry [5 marks]

PRACTICE NOTE: *We intend to ask this very problem with different A3L programs in the midterm*

For this question, consider the exact `A3L` concrete syntax we have used, but assume we have two different implementations of the language : one where every name follows **Static scoping** and another one where every name follows **Dynamic Scoping** consistently throughout their implementations. For our purposes, we may assume the implementations use environments and not substitution, but that is just an implementation detail that does not affect the actual semantics. Both languages are correctly implemented according to their scoping semantices.

Write down the result of evaluating each of the following programs in each implementation.

NOTE: Write one of four things in the boxes:

- If the program evaluates to a number, write the number it evaluates to. E.g., for the program `{+ 1 2}`, write `3`.
- If evaluation of the program runs forever, write the word `forever`.
- If evaluation of the program stops with an error, write the word `error`. You should **not** write the error itself. E.g., for the program `{4 1}`, write `error`.
- If the program evaluates to anything besides these (including closures of any type), write the word `other` in the box. You should **not** write the value itself. E.g., for the program `{fun {x} 1}`, write `other`.

a.

```
{with {f {fun {x} y}}
   {with {y false}
     {+ 12 {whileb y {f 0}}}}}
```

Statically Scoped: [ ] **Dynamically Scoped:** [ ]

b.

```
{with {y x}
  {{fun {x} y} 5}}
```

Statically Scoped: [ ] **Dynamically Scoped:** [ ]

c.

```
{with {f {with {inc 1}
           {fun {arg} {+ arg inc}}}}
  {+ 2 {f 5}}}
```

Statically Scoped: [ ] **Dynamically Scoped:** [ ]

d.

```
{with {f {fun {x} {f x}}}
  {f 0}}
```

Statically Scoped: [ ] **Dynamically Scoped:** [ ]

e.

```
{with {f {with {inc 1}
           {fun {arg} {+ arg inc}}}}
  {with {inc 2}
     {+ inc {f 5}}}}
```

Statically Scoped: [ ] **Dynamically Scoped:** [ ]

2 Just give me all the `with` and `lazy-with`'s you have [9 marks]

PRACTICE NOTE: *We intend to ask a problem in the same style of this one but with* ***different extensions to the language****. We will provide the exact same base EBNF from A3L in the midterm*

The following EBNF is equivalent to the one used in our A3L definition:

```
<A3L> ::= <num> | <boolean> | <id>
        | {<binop> <A3L> <A3L>}
        | {with {<id> <A3L>} <A3L>}
        | {lazy-with {<id> <A3L>} <A3L>}
        | {fun {<id>} <A3L>}
        | {byname-fun {<id>} <A3L>}
        | {<A3L> <A3L>}
        | {ifb <A3L> <A3L> <A3L>}
        | {whileb <A3L> <A3L>}
        | {begin <A3L> <A3Ls>}
        | {setvar! <id> <A3L>}
```

```
<binop> ::= + | - | * | < | =
```

```
<A3Ls> ::=
        | <A3L> <A3Ls>
```

Some eager evaluation languages we had previously discussed had a `with*` expression that can be used to introduce as many mappings as one wanted at once. The A3L sadly had no such feature. For this question, we want to introduce a new syntax for arbitrarily introducing multiple `lazy-with` mappings at once, which we'll call `using`, and will have a peculiar syntax. The following will be valid programs in the syntax we propose:

```
{ {+ x y} using { {{+ 1 2} as x}
                  {{+ x 1} as y}}}

{ {fun {x} x} using { { {+ x y} as y }} }
```

and a program like

```
{ (ifb a b {* 2 5}) using { {{< 1 2} as a}
                            {25 as b}} }
```

is equivalent to the following A3L program:

```
{lazy-with {a {< 1 2}}
  {lazy-with {b 25}
   (ifb a b {* 2 5})}}
```

1. Neatly edit the EBNF to extend the language to include `using` expressions. You may assume that the only newly introduced reserved keywords are `using` and `as`.

   Write your complete answer inlined **in the previous page** where the EBNF has been presented.

2. Extend the definition of the `S-A3L` datatype (before desugaring) to account for `using`. Complete the `s-using` variant in the following space. You may also define other datatypes if you need them.

```
(define-type S-A3L
  [s-num (n number?)]
  ...
  [s-setvar! (var symbol?) (val-expr S-A3L?)]
  [s-using
  
  
  
  
  
  
  ])
```

3. If we decide to implement `using` as a core language feature (instead of implementing it as syntactic sugar) What parts of our intepreter we will **not** need to update? **Mark all the answers that apply**

   - ☐ The `A3L` datatype
   - ☐ The `Env` and `Sto` datatypes
   - ☐ The `Value` datatype
   - ☐ The `parse` function
   - ☐ The `desugar` function
   - ☐ The `interp` function

3 Right-to-Left Store mismanagement (or "Mistakes were made") [11 marks]

PRACTICE NOTE: *We intend to ask a question using this exact language extension, but providing you with a buggy implementation that you will have to fix. You will have to provide the results of running some programs in the original A3L language, the new A3R as specified and* ***also*** *in the buggy implementation.*

Some anonymous language designer has decided to modify the semantics of the A3L language to produce an "A3R" language: In the A3R language, the concrete and abstract syntax are the same as in A3L, but a `binop` evaluates its arguments right-to-left instead of the left-to-right evaluation order used in A3L. This choice, while not a common one, is still a valid design decision.

1. Write down the output value arising from running the following programs in the original A3L language (**A3L** box) and in the specified A3R language (right-to-left) (**Correct A3R** box).

   PRACTICE NOTE: *Remember, in the midterm we'll ask this question with different programs. The midterm will also provide a buggy implementation, and thus each program will also include a* **Buggy A3R** *box to put in the result of running the program in the buggy implementation.*

   NOTE: Follow the same instructions from question 1 on how to fill each of the boxes.

   a.

```
{ - {* 5 10} {+ 8 1}}
```

   A3L: [ ] **Correct A3R:** [ ]

   b.

```
{with {x 5}
  {seqn {setvar! x 1}
        0}}
```

   A3L: [ ] **Correct A3R:** [ ]

   c.

```
{with {a 2}
  {{byname-fun {x} {- x x }}
   {seqn {setvar! a {* a a}}
  a}}}
```

   A3L: [ ] **Correct A3R:** [ ]

   d.

```
{{fun {x} {{fun {y} x} x}} 5}
```

   A3L: [ ] **Correct A3R:** [ ]

   e.

```
{with {f {fun {x} {seqn {setvar! x {+ x 5}}
                        x}}}
  {- {f 2} {f 1}}
```

A3L:  **Correct A3R:** 

3. How can a programmer of the `A3R` language still force that the arguments to a particular `binop` are evaluated left-to-right in their program?

4 Mystery problem [5 marks]

PRACTICE NOTE: *We intend to ask an extra problem in the midterm, of which we do not give you details in advance. This problem will account for 5 marks of the examination.*