1 2F-1 Bookkeeping

- **0 pts** Correct

All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

Exercise 2F-2. Mathematical Induction [5 points]. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof via highlighting or underlining.

Proof: Let F be the set of all flowers and let $\mathsf{smells}(f)$ be the smell of the flower $f \in F$. (The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable. Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) \stackrel{\text{def}}{=} \forall X \in \mathcal{P}(F).\ |X| \leq n \implies (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

One way to formulate the statement to prove is $\forall n \geq 1. P(n)$. We'll prove this by induction on n, as follows:

Base Case: $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction Step: Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same. We will prove that the same thing holds for all subsets of size at most $n+1$. Pick an arbitrary set X such that $|X| = n+1$. Pick two distinct flowers $f, f' \in X$ and let's show that $\mathsf{smells}(f) = \mathsf{smells}(f')$. Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$. Obviously Y and Y' are sets of size at most n so the induction hypothesis holds for both of them. Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" `:-)`)

Exercise 2F-3. While Induction [10 points]. Prove by induction the following statement about the operational semantics:

For any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

Inductive Base Case:
By inversion, we know that we are only dealing with the rules for while. The base case for

2 2F-2 Mathematical Induction

- **0 pts** Correct

All subsequent answers should appear after the first page of your submission and may be shared publicly during peer review.

Exercise 2F-2. Mathematical Induction [5 points]. Find the flaw in the following inductive proof that "All flowers smell the same". Please indicate exactly which sentences are wrong in the proof via highlighting or underlining.

Proof: Let F be the set of all flowers and let $\mathsf{smells}(f)$ be the smell of the flower $f \in F$. (The range of smells is not so important, but we'll assume that it admits equality.) We'll also assume that F is countable. Let the property $P(n)$ mean that all subsets of F of size at most n contain flowers that smell the same.

$$P(n) \stackrel{\text{def}}{=} \forall X \in \mathcal{P}(F).\ |X| \leq n \implies (\forall f, f' \in X.\ \mathsf{smells}(f) = \mathsf{smells}(f'))$$

(the notation $|X|$ denotes the number of elements of X)

One way to formulate the statement to prove is $\forall n \geq 1. P(n)$. We'll prove this by induction on n, as follows:

Base Case: $n = 1$. Obviously all singleton sets of flowers contain flowers that smell the same (by the definition of $P(n)$).

Induction Step: Let n be arbitrary and assume that all subsets of F of size at most n contain flowers that smell the same. We will prove that the same thing holds for all subsets of size at most $n+1$. Pick an arbitrary set X such that $|X| = n+1$. Pick two distinct flowers $f, f' \in X$ and let's show that $\mathsf{smells}(f) = \mathsf{smells}(f')$. Let $Y = X - \{f\}$ and $Y' = X - \{f'\}$. Obviously Y and Y' are sets of size at most n so the induction hypothesis holds for both of them. Pick any arbitrary $x \in Y \cap Y'$. Obviously, $x \neq f$ and $x \neq f'$. We have that $\mathsf{smells}(f') = \mathsf{smells}(x)$ (from the induction hypothesis on Y) and $\mathsf{smells}(f) = \mathsf{smells}(x)$ (from the induction hypothesis on Y'). Hence $\mathsf{smells}(f) = \mathsf{smells}(f')$, which proves the inductive step, and the theorem.

(One indication that the proof might be wrong is the large number of occurrences of the word "obviously" `:-)`)

Exercise 2F-3. While Induction [10 points]. Prove by induction the following statement about the operational semantics:

For any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma \rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even. Make sure you state what you induct on, what the base case is and what the inductive cases are. Show representative cases among the latter. Do not do a proof by mathematical induction!

Inductive Base Case:
By inversion, we know that we are only dealing with the rules for while. The base case for

while is $\langle \texttt{while}\ \mathit{false}\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$. By the recursive definition of while b do c, c is not executed in this case. As such, we see that for this base case, $\sigma' = \sigma$. As such, because $\sigma(x)$ was even, $\sigma'(x)$ must also be even.
We proceed by induction on the structure of the derivation. Assume that the inductive hypothesis holds for all expressions of the form

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

The derivation for this rule is

$$\langle \texttt{while}\ \mathit{true}\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

We proceed one step. Our expression is now

$$\langle \texttt{while}\ \mathit{true}\ \texttt{do}\ (x + 2) := (x + 2) + 2, \sigma\rangle \Downarrow \sigma'$$

Clearly, as x was even, $x+2$ is also even. This completes the inductive step, and we see that for any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even.

Exercise 2F-4. Language Features, Large-Step [12 points]. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{“normal termination”} \\ & | & \sigma\ \texttt{exc}\ n & \text{“exceptional termination”} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma\rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model one type of exception, but every exception has an integer “argument” n (or “payload” or “value”) that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}\ \textsf{seq1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle c_1; c_2, \sigma\rangle \Downarrow t}\ \textsf{seq2}$$

3 2F-3 While Induction

- **0 pts** Correct

while is $\langle \texttt{while}\ \mathit{false}\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$. By the recursive definition of while b do c, c is not executed in this case. As such, we see that for this base case, $\sigma' = \sigma$. As such, because $\sigma(x)$ was even, $\sigma'(x)$ must also be even.
We proceed by induction on the structure of the derivation. Assume that the inductive hypothesis holds for all expressions of the form

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

The derivation for this rule is

$$\langle \texttt{while}\ \mathit{true}\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

We proceed one step. Our expression is now

$$\langle \texttt{while}\ \mathit{true}\ \texttt{do}\ (x + 2) := (x + 2) + 2, \sigma\rangle \Downarrow \sigma'$$

Clearly, as x was even, $x+2$ is also even. This completes the inductive step, and we see that for any BExp b and any initial state σ such that $\sigma(x)$ is even, if

$$\langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$$

then $\sigma'(x)$ is even.

Exercise 2F-4. Language Features, Large-Step [12 points]. We extend IMP with a notion of integer-valued *exceptions* (or *run-time errors*), as in Java, ML or C#. We introduce a new type T to represent command terminations, which can either be normal or exceptional (with an exception value $n \in \mathbb{Z}$):

$$\begin{array}{rcll} T & ::= & \sigma & \text{“normal termination”} \\ & | & \sigma\ \texttt{exc}\ n & \text{“exceptional termination”} \end{array}$$

We use t to range over possible terminations T. We then redefine our operational semantics judgment:

$$\langle c, \sigma\rangle \Downarrow T$$

The interpretation of

$$\langle c, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n$$

is that command c terminated abruptly by throwing an exception with value $n \in \mathbb{Z}$ at a point in c's execution when the state was σ'. We only model one type of exception, but every exception has an integer “argument” n (or “payload” or “value”) that is set when the exception is thrown and available when the exception is caught.

Note that our previous command rules must be updated to account for exceptions, as in:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}{\langle c_1; c_2, \sigma\rangle \Downarrow \sigma'\ \texttt{exc}\ n}\ \textsf{seq1} \qquad \frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \quad \langle c_2, \sigma'\rangle \Downarrow t}{\langle c_1; c_2, \sigma\rangle \Downarrow t}\ \textsf{seq2}$$

We also introduce three additional commands:

$$\begin{array}{l}\mathsf{throw}\ e\\ \mathsf{try}\ c_1\ \mathsf{catch}\ x\ c_2\\ \mathsf{after}\ c_1\ \mathsf{finally}\ c_2\end{array}$$

- The throw e command raises an exception with argument e.
- The try command executes c_1. If c_1 terminates normally (i.e., without an uncaught exception), the try command also terminates normally. If c_1 raises an exception with value e, the variable $x \in L$ is assigned the value e and then c_2 is executed.
- The finally command executes c_1. If c_1 terminates normally, the finally command terminates by executing c_2. If instead c_1 raises an exception with value e_1, then c_2 is executed:
  - If c_2 terminates normally, the finally command terminates by throwing an exception with value e_1. (That is, the original exception e_1 is re-thrown at the end of the finally block, as in Java.)
  - If c_2 throws an exception with value e_2, the finally command terminates by throwing an exception with value e_2. (That is, the new exception e_2 overrides the original exception e_1, also as in Java.)

These constructs are intended to have the standard exception semantics from languages like Java, C# or OCaml — except that the catch block merely assigns to x, it does not bind it to a local scope. So unlike Java, our catch does not behave like a let. We thus expect:

```
x := 0 ;
{ try
    if x <= 5 then throw 33 else throw 55
  catch x
    print x } ;
while true do {
  x := x - 15 ;
  print x ;
  if x <= 0 then throw (x*2) else skip
}
```

to output "`33 18 3 -12`" and then terminate with an uncaught exception with value `-24`.

Give the large-step operational semantics inference rules (using our new judgment) for the three new commands presented here. You should present six (6) new rules total.

For the throw command, we have a very straightforward rule, as we know there will be an exception raised. Thus, the rule is:

$$\frac{< e, \sigma > \Downarrow n}{< \text{throw } e, \sigma > \Downarrow \sigma \text{ exc } n}$$

For try catch, we must consider the case where c_1 raises an exception as well as the case where it doesn't. For the first case, only c_1 will be executed. Thus, the rule is

$$\frac{< c_1, \sigma >\Downarrow \sigma'}{< \text{try } c_1 \text{ catch } x \ c_2, \sigma >\Downarrow \sigma'}$$

The alternative case is when c_1 raises an exception. As such, we need to assign x to the value of that expression, and rely on c_2's termination to complete the rule. We also must assign x to the value of the exception before executing c_2. Thus, we have:

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n \qquad < x := n; c_2, \sigma' >\Downarrow t}{< \text{try } c_1 \text{ catch } x \ c_2, \sigma >\Downarrow t}$$

Accordingly, for an after...finally statement, we will need three cases. The first is where c_1 terminates normally. This is straightforward, and will rely on the termination of c_2 to complete the rule.

$$\frac{< c_1, \sigma >\Downarrow \sigma' \qquad < c_2, \sigma' >\Downarrow t}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow t}$$

The second case we must consider is when c_1 raises an exception, but c_2 terminates normally. In this case, the statement will terminate with the same exception that c_1 raises. Thus, the rule is

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n \qquad < c_2, \sigma' >\Downarrow \sigma''}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow \sigma'' \text{ exc } n}$$

The final case is similar to the second, only c_2 now also raises an exception, and this is the one that gets re-thrown at the end of execution. Therefore, the rule is:

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n_1 \qquad < c_2, \sigma' >\Downarrow \sigma'' \text{ exc } n_2}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow \sigma'' \text{ exc } n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

I disagree with the idea that describing "IMP with exceptions" using small-step semantics would be more elegant. The reason for this is primarily that it abstracts away the need to define (try catch) and (after finally) in terms of the implicit throw operation that is done whenever one of the constituent commands raises an exception. The reason for this necessity is that throw n is a terminal program, and would need to be accounted for in the redexes and local reduction rules. With large-step semantics, this is not necessary, as we can assume for a given case that if an exception is raised, the throw has already been resolved, and we can just focus on what needs to happen with the termination of the command as a whole.

4 2F-4 Language Features, Large Step

- **0 pts** Correct

For try catch, we must consider the case where c_1 raises an exception as well as the case where it doesn't. For the first case, only c_1 will be executed. Thus, the rule is

$$\frac{< c_1, \sigma >\Downarrow \sigma'}{< \text{try } c_1 \text{ catch } x \ c_2, \sigma >\Downarrow \sigma'}$$

The alternative case is when c_1 raises an exception. As such, we need to assign x to the value of that expression, and rely on c_2's termination to complete the rule. We also must assign x to the value of the exception before executing c_2. Thus, we have:

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n \qquad < x := n; c_2, \sigma' >\Downarrow t}{< \text{try } c_1 \text{ catch } x \ c_2, \sigma >\Downarrow t}$$

Accordingly, for an after...finally statement, we will need three cases. The first is where c_1 terminates normally. This is straightforward, and will rely on the termination of c_2 to complete the rule.

$$\frac{< c_1, \sigma >\Downarrow \sigma' \qquad < c_2, \sigma' >\Downarrow t}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow t}$$

The second case we must consider is when c_1 raises an exception, but c_2 terminates normally. In this case, the statement will terminate with the same exception that c_1 raises. Thus, the rule is

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n \qquad < c_2, \sigma' >\Downarrow \sigma''}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow \sigma'' \text{ exc } n}$$

The final case is similar to the second, only c_2 now also raises an exception, and this is the one that gets re-thrown at the end of execution. Therefore, the rule is:

$$\frac{< c_1, \sigma >\Downarrow \sigma' \text{ exc } n_1 \qquad < c_2, \sigma' >\Downarrow \sigma'' \text{ exc } n_2}{< \text{after } c_1 \text{ finally } c_2, \sigma >\Downarrow \sigma'' \text{ exc } n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. Argue for or against the claim that it would be more natural to describe "IMP with exceptions" using small-step contextual semantics. You may use "simpler" or "more elegant" instead of "more natural" if you prefer. Do not exceed two paragraphs (one should be sufficient). Both your ideas and also the clarity with which they are expressed (i.e., your English prose) matter.

I disagree with the idea that describing "IMP with exceptions" using small-step semantics would be more elegant. The reason for this is primarily that it abstracts away the need to define (try catch) and (after finally) in terms of the implicit throw operation that is done whenever one of the constituent commands raises an exception. The reason for this necessity is that throw n is a terminal program, and would need to be accounted for in the redexes and local reduction rules. With large-step semantics, this is not necessary, as we can assume for a given case that if an exception is raised, the throw has already been resolved, and we can just focus on what needs to happen with the termination of the command as a whole.

Exercise 2C. Language Features, Coding. Download the Homework 2 code pack from the course web page. Modify `hw2.ml` so that it implements a complete interpreter for "IMP with exceptions (and `print`)". You may build on your code from Homework 1 (although the `let` command is not part of this assignment). Using OCaml's exception mechanism to implement IMP exceptions is actually slightly harder than doing it "naturally", so I recommend that you just implement the operational semantics rules. The `Makefile` includes a "`make test`" target that you should use (at least) to test your work.

Hint: to check if a termination `term` is an exception, use syntax like

```
begin match term with
| Normal -> do_something
| Exceptional(n) -> do_something_else using n
end
```

Modify the file `example-imp-command` so that it contains a "tricky" terminating IMP command (presumably involving exceptions) that can be parsed by our IMP test harness (e.g., "`imp < example-imp-command`" should not yield a parse error).

Submission. Turn in the formal component of the assignment as a single PDF document via the `gradescope` website. Your name and Michigan email address must appear on the first page of your PDF submission but may not appear anywhere else. Turn in the coding component of the assignment via the `autograder.io` website.

5 2F-5 Language Features, Analysis

- **0 pts** Correct