1 2F-1 Bookkeeping

- **0 pts** Correct

**Peer Review ID: 65783874** — enter this when you fill out your peer evaluation via gradescope

**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.

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

(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" `:-)`)

**Answer** The answer is that the inductive step is ill-defined. It is necessary, for n = 2, to have $Y$ and $Y'$ such that the intersection of $Y \cap Y'$ is empty; there are no items in there for x to refer to, and a set of 0 items is below our base case and as such has an indeterminate smell. With an invalid inductive step, this proof does not work.

**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 $\sigma$ 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!

## 2 2F-2 Mathematical Induction

**- 0 pts** Correct

Peer Review ID: 65783874 — enter this when you fill out your peer evaluation via gradescope

**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 $\mathsf{smells}$ 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.

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

(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" `:-)`)

**Answer** The answer is that the inductive step is ill-defined. It is necessary, for n = 2, to have $Y$ and $Y'$ such that the intersection of $Y \cap Y'$ is empty; there are no items in there for x to refer to, and a set of 0 items is below our base case and as such has an indeterminate smell. With an invalid inductive step, this proof does not work.

**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 $\sigma$ 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!

**Answer** This proof cannot be done with mathematical induction on x, as there is no minimal base case with integers. Instead, I use structural induction on number of loops of the "while" command. The substructures of this command are each loop: how the command behaves when the Bexp b is true, and when it is false. Let us define one base case, representing the break case of the while loop, causing the command to produce a $\sigma'$. This is case b == false; if this is the case, then the while command's subcommands do not run. $\sigma == \sigma'$; since $\sigma(x)$ is even, $\sigma'(x)$ is the same and is even as well.
The inductive cases are each substructure of the while loop; the number of times b is true and c1 (x := x + 2) is executed. If b == true is the case, then the command x := x + 2 executes. We know that for all x if x is even, x + 2 is even (as per Piazza, we are not required to prove this). When the command executes, we transition from a state $\sigma$ where x is even to a state $\sigma'$ where x is even. These are the only two possible states for iterations of the while loop.
At this point, we apply induction over the while loops. If $\sigma(x)$ is even, and x := x + 2 executes n times, $\sigma'(x)$ will be even as well. If the loop executes again, a n+1th time, $\sigma'(x)$ will become $\sigma''(x)$ and is guaranteed to be even as well. By the inductive hypothesis, this is true for any number of substructure iterations; both b == false and b == true evaluate to $\sigma'(x)$ as even. Thus, for any initial state where $\sigma(x)$ is even, if the while command is run and evaluates to a subsequent $\sigma'$, $\sigma'(x)$ 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 $\sigma'$. 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

**Answer** This proof cannot be done with mathematical induction on x, as there is no minimal base case with integers. Instead, I use structural induction on number of loops of the "while" command. The substructures of this command are each loop: how the command behaves when the Bexp b is true, and when it is false. Let us define one base case, representing the break case of the while loop, causing the command to produce a $\sigma'$. This is case b == false; if this is the case, then the while command's subcommands do not run. $\sigma == \sigma'$; since $\sigma(x)$ is even, $\sigma'(x)$ is the same and is even as well.
The inductive cases are each substructure of the while loop; the number of times b is true and c1 (x := x + 2) is executed. If b == true is the case, then the command x := x + 2 executes. We know that for all x if x is even, x + 2 is even (as per Piazza, we are not required to prove this). When the command executes, we transition from a state $\sigma$ where x is even to a state $\sigma'$ where x is even. These are the only two possible states for iterations of the while loop.
At this point, we apply induction over the while loops. If $\sigma(x)$ is even, and x := x + 2 executes n times, $\sigma'(x)$ will be even as well. If the loop executes again, a n+1th time, $\sigma'(x)$ will become $\sigma''(x)$ and is guaranteed to be even as well. By the inductive hypothesis, this is true for any number of substructure iterations; both b == false and b == true evaluate to $\sigma'(x)$ as even. Thus, for any initial state where $\sigma(x)$ is even, if the while command is run and evaluates to a subsequent $\sigma'$, $\sigma'(x)$ 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 $n \in \mathbb{Z}$):

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

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 $\sigma'$. 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:

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

- 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.

**Answers** We need one rule per ”case”–as there is one case for throw, two for try, and three for finally it makes sense to assign the new rules in this way. Throw has the simplest rule:

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle \text{throw e}, \sigma \rangle \Downarrow \sigma' \ \texttt{exc}\ n}$$

. We define a rule for try if an exception is thrown, and one if it is not:

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{try c1 catch x c2}, \sigma\rangle \Downarrow t[x := n]}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{try c1 catch x c2}, \sigma\rangle \Downarrow \sigma'}$$

Finally, we define three rules for after/finally; one if c1 terminates normally, one if c1 raises an exception but c2 does not, and one if c1 and c2 both raise exceptions.

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow t}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n \langle c2, \sigma'\rangle \Downarrow \sigma''}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow \sigma'' \texttt{ exc } n}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n1 \langle c2, \sigma'\rangle \Downarrow \sigma'' \texttt{ exc } n2}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow \sigma'' \texttt{ exc } n2}$$

**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.

**Answer** I argue that it would be much less natural to describe"IMP with exceptions" using small-step contextual semantics. Small-step contextual semantics involve atomic steps, with atomic rewrites every step. Atomic reductions are performed in a context H. However, exceptions "clear the board", so to speak. While small-step semantics do have state, there would need to be rules that if an exception is thrown by a command every subsequent command would be reduced to skip until the exception is handled (by a finally or catch block, or the program ending). This would complicate the semantic rules heavily–small-step semantics are not strictly ordered, but solved in terms of open redexes, and (for example) modifying state in an operation which happens after an exception is called would be very problematic.

**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

## 4 2F-4 Language Features, Large Step

**- 0 pts** Correct

. We define a rule for try if an exception is thrown, and one if it is not:

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{try c1 catch x c2}, \sigma\rangle \Downarrow t[x := n]}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{try c1 catch x c2}, \sigma\rangle \Downarrow \sigma'}$$

Finally, we define three rules for after/finally; one if c1 terminates normally, one if c1 raises an exception but c2 does not, and one if c1 and c2 both raise exceptions.

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \langle c2, \sigma'\rangle \Downarrow t}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow t}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n \langle c2, \sigma'\rangle \Downarrow \sigma''}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow \sigma'' \texttt{ exc } n}$$

$$\frac{\langle c1, \sigma\rangle \Downarrow \sigma' \texttt{ exc } n1 \langle c2, \sigma'\rangle \Downarrow \sigma'' \texttt{ exc } n2}{\langle \text{after c1 finally c2}, \sigma\rangle \Downarrow \sigma'' \texttt{ exc } n2}$$

**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.

**Answer** I argue that it would be much less natural to describe"IMP with exceptions" using small-step contextual semantics. Small-step contextual semantics involve atomic steps, with atomic rewrites every step. Atomic reductions are performed in a context H. However, exceptions "clear the board", so to speak. While small-step semantics do have state, there would need to be rules that if an exception is thrown by a command every subsequent command would be reduced to skip until the exception is handled (by a finally or catch block, or the program ending). This would complicate the semantic rules heavily–small-step semantics are not strictly ordered, but solved in terms of open redexes, and (for example) modifying state in an operation which happens after an exception is called would be very problematic.

**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

## 5 2F-5 Language Features, Analysis

**- 0 pts** Correct