1 2F-1 Bookkeeping

- 0 pts Correct

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" `:-)`)

Remark: This proof breaks down when picking aribtirary $x \in Y \cap Y'$ because such an x may not exist. As such, we cannot apply the inductive hypothesis to Y or Y' and the inductive step, so the theorem has not been proven true.

(Indeed, though we have both $P(1)$ and $\forall n \geq 2. P(n) \to P(n+1)$, it is not the case that $P(1) \to P(2)$; we could easily find a counterexample of two distinct flowers smelling different to prove the "theorem" false.)

2 2F-2 Mathematical Induction

- 0 pts Correct

Exercise 2F-3. While Induction [10 points]. We want to prove that

$$\langle \text{while } b \text{ do } x := x \ + \ 2, \sigma\rangle \Downarrow \sigma' \implies \forall b \in \text{BExp}.\forall \sigma \in \Sigma. \text{ initial state } \sigma \text{ s.t. } \sigma(x) \text{ is even} \implies \sigma'(x) \text{ is even}$$

We let x, σ such that x is intially even, σ', and $D :: \langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma'$ be arbitrary.
We proceed by induction on the structure of the derivation D. Our inductive hypothesis assumes the desired property holds for subderivations of D.

By inversion, the last rule used in the derivation D must be `while false` or `while true`. The base case is where only one rule is ever applied to get D: `while false`. We never execute the loop body in this case, so $\sigma'(x) = \sigma(x)$, meaning that if $\sigma(x)$ is even, then $\sigma'(x)$ is certainly even as well.

The inductive cases have some number of applications of the `while true` rule. We first consider the inductive case where the last rule used in D is `while false`:

- By inversion, $D_1 :: \langle b, \sigma_1\rangle \Downarrow$ false, so the loop body is not executed and σ_1 is unchanged. By the inductive hypothesis on $D2 :: \langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma_1$, if D_2 had initial state σ such that $\sigma(x)$ was even, then D_2's ending state σ_1 must also have $\sigma_1(x)$ even. As D_1 does nothing to the state σ_1 when the last rule is `while false`, D's ending state $\sigma' = \sigma_1$, so $\sigma'(x)$ is even as well.

We then consider the inductive case where the last rule used in D is `while true`:

- By inversion, $D_1 :: \langle b, \sigma_1\rangle \Downarrow$ true, $D_2 :: \langle x := x + 2, \sigma_1\rangle \Downarrow \sigma_2$, and $D_3 :: \langle \texttt{while b do x := x + 2}, \sigma_2\rangle \Downarrow \sigma'$. By the inductive hypothesis on $D4 :: \langle \texttt{while}\ b\ \texttt{do}\ x := x + 2, \sigma\rangle \Downarrow \sigma_1$, if D_4 had initial state σ such that $\sigma(x)$ was even, then D_4's ending state σ_1 must also have $\sigma_1(x)$ even. D_2 adding 2 to an even number $\sigma_1(x)$, which maintains its parity, so $\sigma_2(x)$ is even as well. By the inductive hypothesis on D_3, since $\sigma_2(x)$ is even, it follows that $\sigma'(x)$ is also even.

The above rules comprise all possible boolean expressions as every boolean expression must evaluate to true or false. As we've shown via structural induction on D, a while loop whose c only adds 2 to a starting value x preserves x's even parity.

3 2F-3 While Induction

- 0 pts Correct

Exercise 2F-4. Language Features, Large-Step [12 points].
`throw`ing an arithmetic expression leads to terminating in an exception state where the evaluation of that expression is raised:

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

`try`ing a command that terminates normally just executes that command:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \text{try } c_1 \text{ catch x } c_2, \sigma \rangle \Downarrow \sigma'}$$

`try`ing a command that raises an exception leads to storing that exception value in memory, then executing the (possibly good, possibly exception-raising) command c_2:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{ exc } n \quad \langle x := n, \sigma' \rangle \Downarrow \sigma'[x := n] \quad \langle c_2, \sigma'[x := n] \rangle \Downarrow t}{\langle \text{try } c_1 \text{ catch x } c_2, \sigma \rangle \Downarrow t}$$

if c_1 terminates normally, `finally` just executes c_2 à la the `seq2` command:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow t}{\langle \text{after } c_1 \text{ finally } c_2, \sigma \rangle \Downarrow t}$$

if c_1 terminates with an exception n_1 and c_2 terminates normally, `finally` re-throws n_1 at the end of the `finally` block:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \text{ exc } n_1 \quad \langle c_2, \sigma' \rangle \Downarrow \sigma'' \quad \langle \text{throw } n_1, \sigma'' \rangle \Downarrow \sigma'' \text{ exc } n_1}{\langle \text{after } c_1 \text{ finally } c_2, \sigma \rangle \Downarrow \sigma'' \text{ exc } n_1}$$

if c_1 terminates with an exception n_1 and c_2 terminates with an exception n_2, `finally` lets n_2 be thrown:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \text{ exc } n_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \text{ exc } n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. I posit that using small-step contextual semantics would be a more natural way of describing "IMP with exceptions" than large-step contextual semantics. As exceptions are largely used for error-handling and control flow where the details of which steps are taken when are important, expressing these details in redexes that are atomitcally reduced within contexts is more useful for reasoning about the intricacies of a langugage's exception rules. Instead of having to update the previous large-step command rules to account for exceptions, we can just add some redexes and reduction rules to capture the additional behavior that's now possible in IMP. Adding rules when adding new functionality seems more natural and elegant than both adding new rules and modifying old ones. Further, redexes could more elegant express things like a throw statement causing a while loop to be exited abrutly and seem to more naturally account for all the types of behavior potentially introduced by exceptions.

Exercise 2C. Language Features, Coding. See submission at `autograder.io`

4 2F-4 Language Features, Large Step

- **0 pts** Correct

if c_1 terminates with an exception n_1 and c_2 terminates with an exception n_2, `finally` lets n_2 be thrown:

$$\frac{\langle c_1, \sigma\rangle \Downarrow \sigma' \text{ exc } n_1 \quad \langle c_2, \sigma'\rangle \Downarrow \sigma'' \text{ exc } n_2}{\langle \text{after } c_1 \text{ finally } c_2, \sigma\rangle \Downarrow \sigma'' \text{ exc } n_2}$$

Exercise 2F-4. Language Features, Analysis [6 points]. I posit that using small-step contextual semantics would be a more natural way of describing "IMP with exceptions" than large-step contextual semantics. As exceptions are largely used for error-handling and control flow where the details of which steps are taken when are important, expressing these details in redexes that are atomitcally reduced within contexts is more useful for reasoning about the intricacies of a langugage's exception rules. Instead of having to update the previous large-step command rules to account for exceptions, we can just add some redexes and reduction rules to capture the additional behavior that's now possible in IMP. Adding rules when adding new functionality seems more natural and elegant than both adding new rules and modifying old ones. Further, redexes could more elegant express things like a throw statement causing a while loop to be exited abrutly and seem to more naturally account for all the types of behavior potentially introduced by exceptions.

Exercise 2C. Language Features, Coding. See submission at `autograder.io`

5 2F-5 Language Features, Analysis

- **0 pts** Correct