Runtime Errors

- Like divergence, we use a pseudo-state (also a pseudo-value) "$\perp_e$" to represent runtime error.
- Runtime errors can happen in both expressions and programs. Let's look at runtime errors in expressions first. We write $\sigma(e) = \perp_e$ to represent that the evaluation of expression e in state σ causes a runtime error.
  - Here "$\perp_e$" is a pseudo-value, which means it is not a real value, it is used to represent values such as $x/0$ or $sqrt(-1)$.
  - If e might cause runtime error, then instead of $\sigma(e) \in V$ for some set value set V, we now have $\sigma(e) \in V \cup \{\perp_e\}$ as the range of $\sigma(e)$. For example, since x/y might cause a runtime error, then its value in some state σ might result in $\perp_e$, thus we write $\sigma(x/y) \in \mathbb{Z} \cup \{\perp_e\}$.

- Types of runtime errors:
  - Primary Failures: The primitive values and operations being supported determine some set of basic runtime errors.
    - Array index out of bounds: $\sigma(b[e]) = \perp_e$ if $\sigma(e) < 0$ or $\sigma(e) \geq \sigma\big(size(b)\big)$ Similar situation for multi-dimensional arrays.
    - Division by zero: $\sigma(e_1/e_2) = \sigma(e_1 \,\%\, e_2) = \perp_e$ if $\sigma(e_2) = 0$.
    - Square root of negative number: $\sigma\left(sqrt(e)\right) = \perp_e$ if $\sigma(e) < 0$.

  - Hereditary Failure: If evaluating a subexpression fails, then the overall expression fails.
    - If op is a unary operator, then $\sigma(op\ e) = \perp_e$ if $\sigma(e) = \perp_e$. For example: $e \equiv x/0$, then $\sigma(e) = \perp_e$, and $\sigma\big(sqrt(e)\big) = \perp_e$ as well.
    - If op is a binary operator, then $\sigma(e_1\ op\ e_2) = \perp_e$ if $\sigma(e_1) = \perp_e$ or $\sigma(e_2) = \perp_e$.
    - For a conditional expression, $\sigma(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) = \perp_e$ if one of the following three situations occurs:
      1) $\sigma(B) = \perp_e$
      2) $\sigma(B) = T$ and $\sigma(e_1) = \perp_e$
      3) $\sigma(B) = F$ and $\sigma(e_2) = \perp_e$

1. What are the states that will cause runtime errors for each of the following expressions?
   a. $b[x/y]$ Some state σ where $\sigma(y) = 0$, or $\sigma(x/y) < 0$ or $\sigma(x/y) \geq \sigma\big(size(b)\big)$.
   b. $sqrt(x) + sqrt(x/y)$ Some state σ where $\sigma(x) < 0$ or $\sigma(y) = 0$ or $\sigma(x/y) < 0$.
   c. $\textbf{if } y = 0 \textbf{ then } 0 \textbf{ else } x/y \textbf{ fi}$ Never causes runtime error.

- A runtime error in an expression can cause the statement it appears in to halt unsuccessfully. We write $\langle S, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$ for the operational semantics of such a statement.
  - Here $\perp_e$ is a pseudo-state not a pseudo-value. Decide whether $\perp_e$ is a pseudo-state or pseudo-value from the context.

- With runtime errors, let's expand our operations semantics rules:
  - $\sigma(e) = \perp_e$, then $\langle v \coloneqq e, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.
  - $\sigma(b[e_1]) = \perp_e$ or $\sigma(e_2) = \perp_e$, then $\langle b[e_1] \coloneqq e_2, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.
  - If $\sigma(B) = \perp_e$, then $\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.
    - On the other hand, if $\sigma(B) \neq \perp_e$, we will continue to evaluate configuration $\langle S_1, \sigma\rangle$ or $\langle S_2, \sigma\rangle$.
  - If $\langle S_1, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$ then $\langle S_1; S_2, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.

- If $\sigma(B) = \perp_e$, then $\langle \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle \rightarrow \langle E, \perp_e\rangle$.
  - On the other hand, if $\sigma(B) \neq \perp_e$, we will continue to evaluate configuration $\langle E, \sigma\rangle$ or $\langle S; \textbf{while } B \textbf{ do } S \textbf{ od}, \sigma\rangle$.

2. Evaluate $S \equiv x := 0; y := 1; W$ where $W \equiv \textbf{while } x/y \geq 0 \textbf{ do } y := sqrt(y) - 1 \textbf{ od}$ in state σ.

$\langle S, \sigma\rangle \rightarrow^* \langle W, \sigma[x \mapsto 0][y \mapsto 1]\rangle$
$\rightarrow^* \langle W, \sigma[x \mapsto 0][y \mapsto 0]\rangle$ //since $\sigma[x \mapsto 0][y \mapsto 1](x/y \geq 0) = T$
$\rightarrow \langle E, \perp_e\rangle$ //since $\sigma[x \mapsto 0][y \mapsto 0](x/y \geq 0) = \perp_e$

(Properties and consequences of ⊥)

- ⊥ refers generically to $\perp_d$ and/or $\perp_e$. We use $\langle S, \sigma\rangle \rightarrow^* \langle E, \perp\rangle$ when it's not important which of $\perp_e$ or $\perp_d$ can occur. Similarly, $\perp \in M(S, \sigma)$ means $\langle S, \sigma\rangle$ leads to $\perp_d$ or $\perp_e$.

- Since we use ⊥ somewhere an actual memory state appears in evaluating a program, we want to look at other situations where a state appears in and think about whether ⊥ can be used there.
  - ⊥ is not a well-formed state.
  - When we say, "for all states…", "for some state…", or "for all semantic values…", "for some semantic values…", we don't include ⊥.
  - We cannot add a binding to ⊥: $\perp \cup \{v = \beta\} = \perp$.
  - Consider⊥ as a pseudo-value, then binding it with a variable will result in a pseudo-state: $\sigma(v) \neq \perp$ and $\sigma[v \mapsto \perp] = \perp$. In other words, we cannot bind a variable with pseudo-value ⊥.
  - Evaluating a variable or an expression in ⊥ results in a pseudo-value ⊥: If $\sigma = \perp$ then $\sigma(v) = \sigma(e) = \perp$. In other words, we cannot take the value of a variable or expression in ⊥.
  - Operationally, execution halts as soon we generate ⊥ as a pseudo-state: $\langle S, \perp\rangle \rightarrow^0 \langle E, \perp\rangle$.
  - Denotationally, we can't run a program in pseudo-state ⊥: $M(S, \perp) = \{\perp\}$.
  - We can't evaluate something after generating ⊥.

- Logic with ⊥
  - ⊥ cannot satisfy any predicate: $\perp \nvDash p$ for all p, even if p is the constant T. In general, we now have three possibilities for a state trying to satisfy a predicate: $\sigma \vDash p$, $\sigma \vDash \neg p$, or $\sigma(p) = \perp$.
    - Previously we have $\sigma \nvDash p$ means $\sigma \vDash \neg p$, but this is no longer true when ⊥ is taken into consideration. $\sigma \nvDash p$ means "It is not true that $\sigma \vDash p$" and it means $\sigma \vDash \neg p$ or $\sigma(p) = \perp$ (in other words, $\sigma \nvDash p \Leftrightarrow \sigma \vDash \neg p \vee \sigma(p) = \perp$) now.

3. Let $e \equiv \textbf{if } x = 0 \textbf{ then } x / x = 0 \textbf{ else } 2 = 2 \textbf{ fi}$.

a. Is e a legal expression? Yes, it has Booleans value.
b. If $\sigma = \{x = 0\}$ proper for e? Yes.
c. Does $\sigma = \{x = 0\}$ satisfy e? No, $\sigma(x / x = 0) = \perp$.
d. Does $\sigma = \{x = 1\}$ satisfy $\neg e$? No, $\neg(2 = 2) = F$.

4. True or False.

a. If $\sigma \vDash p$, then $\sigma \nvDash \neg p$. True.
b. If $\sigma \vDash p$, then $\sigma(p) \neq \perp$. True.
c. If $\sigma \vDash \neg p$, then $\sigma \nvDash p$. True.
d. If $\sigma \nvDash \neg p$, then $\sigma \vDash p$. False.
e. If $\sigma(p) = \perp$, then $\sigma(\neg p) = \perp$. True.
f. If $\sigma(p) \neq \perp$, then $\sigma \nvDash p \Leftrightarrow \sigma \vDash \neg p$ True.

g. If p is a valid predicate (in other words, $\vDash p$), then $\perp \vDash p$. False.

h. If $\nvDash p$, then there exists some σ with $\sigma(p) = \perp$. False. It means $\exists \sigma. \sigma(p) = \perp \vee \sigma(p) = F$

Nondeterministic Program

- Nondeterminism is a theoretical idea, in general it means "don't make decision, consider all possible outcomes at the same time with same probability." Note that, it doesn't mean "randomly pick possible outcome". Just like Schrodinger' cat, it is both alive and dead at the same time with a 50% 50% probability.
  - For example, when we say, "choose one side of a coin nondeterministically", it means "choose head with 50% probability and choose tail with 50% probability." It doesn't "pick one side randomly" because it will be either a head or a tail in this way.
    - As an aside, a nondeterministic program can be simulated by "random picking a branch among all possible branches" a lot of times. For example, you can simulate the procedure "choosing one side of a coin nondeterministically" by tossing a fair coin a lot of times, then you will come up with a set of outputs:$\{head,\ tail\}$, and the probability of two outputs will be near 50% and 50%.

  - Here is another example. If we have a simple program that returns the maximum between x and y:

    $$\textbf{if } x \leq y \textbf{ then } max \coloneqq y \textbf{ else } max \coloneqq x \textbf{ fi}$$

    This $\textbf{if} - \textbf{else}$ statement is deterministic. When we have $x = y$, it will always choice y to be the max. Consider a different program that does the following:

    "If $x \geq y$ then $max \coloneqq x$, if $x \leq y$ then $max \coloneqq y$"

    and it can evaluate both branches at the same time. When we have $x = y$, then max should have 50% chance to be x and 50% chance to be y.

- Actually, a machine/program cannot run *nondeterministically*, but designing a nondeterministic machine/program is useful:
  1) A nondeterministic machine/program has a deterministic machine/program that can do the same job (although the deterministic version might need to finish the same job using much longer time). This is a topic in CS530 Computational Theory.
  2) The design process can be simplified. We can avoid some insignificant choice-making and only focus on the important decisions in the design. The design of a nondeterministic machine is easier to read and understand.

(Syntax of Nondeterministic Statements in our Programming language)

- A nondeterministic $\textbf{if} - \textbf{fi}$ statement: $\textbf{if } B_1 \to S_1 \ \square \ B_2 \to S_2 \ \square \ \ldots \square \ B_n \to S_n \ \textbf{fi}$
  - We use box symbols to separate different arms.
  - Inside each arm we have a **guarded command** $B_i \to S_i$ which means "if **guard** B_i is true then it's okay to run S_i".
  - How it works:
    - If none of the guards B_i is true, it aborts with a **runtime error.**
    - If exactly one guard B_i is true, then it executes S_i.
    - If more than one guard is true, then ***choose one*** of the true arms ***nondeterministically*** and execute the corresponding S_i.
    - Here, "***choose one… nondeterministically***" means, the machine splits into several copies of itself, and each copy follows one of the possible arms in parallel; and each copy holds a probability so that the total probability adds up to 1.

5. Write a nondeterministic **if** $-$ **fi** statement that chooses the maximum between x and y.

$$\textbf{if } x \geq y \rightarrow max := x \ \square \ x \leq y \rightarrow max := y \textbf{ fi}$$

6. Write a nondeterministic **if** $-$ **fi** statement to simulate **if** B **then** S_1 **else** S_2 **fi**

$$\textbf{if } B \rightarrow S_1 \ \square \ \neg B \rightarrow S_2 \textbf{ fi}$$

- We express a nondeterministic **while** loop using a nondeterministic **do** $-$ **od** statement: **do** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \square \ B_n \rightarrow S_n$ **do**
  - How it works:
    - At the beginning of each iteration, check all guards.
    - If none of the guards B_i is true, then finish loop.
    - If exactly one B_i is true, then execute S_i and go to the next iteration.
    - If more than one guard is true, then choose one of the true arms *nondeterministically* and execute the corresponding S_i; then go to the next iteration.

(Operational Semantics of Nondeterministic Program)

- For **if** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \square \ B_n \rightarrow S_n$ **fi.**
  - Let $IF \equiv$ **if** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \square \ B_n \rightarrow S_n$ **fi** and $BB \equiv B_1 \vee B_2 \vee \ldots \vee B_n$.
    - If $\sigma(BB) = \perp$, then $\langle IF, \sigma \rangle \rightarrow \langle E, \perp \rangle$.
    - If $\sigma(BB) = F$, then $\langle IF, \sigma \rangle \rightarrow \langle E, \perp_e \rangle$.
    - If there is exactly one guard with $\sigma(B_i) = T$, then $\langle IF, \sigma \rangle \rightarrow \langle S_i, \sigma \rangle$.
    - If there is more than one guard with $\sigma(B_i) = T$, then choose one of those i *nondeterministically* and jump to that S_i: $\langle IF, \sigma \rangle \rightarrow \langle S_i, \sigma \rangle$.

7. Evaluate $S \equiv$ **if** $x \geq 0 \rightarrow x := x * x \ \square \ x \leq 8 \rightarrow x := -x$ **fi** in $\{x = 3\}$.
   Since both guards are true, so $\langle S, \{x = 3\} \rangle$ can *nondeterministically* go to one of $\langle x := x * x, \{x = 3\} \rangle \rightarrow \langle E, \{x = 9\} \rangle$ and $\langle x := -x, \{x = 3\} \rangle \rightarrow \langle E, \{x = -3\} \rangle$.

- For **do** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \square \ B_n \rightarrow S_n$ **od.**
  - Let $DO \equiv$ **do** $B_1 \rightarrow S_1 \ \square \ B_2 \rightarrow S_2 \ \square \ \ldots \square \ B_n \rightarrow S_n$ **od** and $BB \equiv B_1 \vee B_2 \vee \ldots \vee B_n$.
    - If $\sigma(BB) = \perp$, then $\langle DO, \sigma \rangle \rightarrow \langle E, \perp \rangle$.
    - If $\sigma(BB) = F$, then $\langle DO, \sigma \rangle \rightarrow \langle E, \sigma \rangle$.
    - If there is exactly one guard with $\sigma(B_i) = T$, then $\langle DO, \sigma \rangle \rightarrow \langle S_i; DO, \sigma \rangle$.
    - If there is more than one guard with $\sigma(B_i) = T$, then choose one of those i *nondeterministically* and jump to that S_i: $\langle DO, \sigma \rangle \rightarrow \langle S_i; DO, \sigma \rangle$.