Predicate functions

- We like to give names to predicates and parameterize them, in other words, we like to write predicates into functions from a set of variables to a Boolean value. These functions are called predicate functions.
  - One reason for using predicate functions is that it can simplify our program, so it is easier to read.
    - For example, in our program we need to check "whether integer x is the square of integer r" constantly. Then we can write a predicate "$x \geq 0 \wedge r * r = x$". If we define $isSq(x, r) \equiv x \geq 0 \wedge r * r = x$, then we can simply call $isSq(x, r)$ whenever we need to check.

  - Another reason is that predicate functions help us to handle expressions with expanded domains of variables.
    - For example, if we define $L(x, y) \equiv$ "x loves y", where the domain of x and y are all people, then we can express "Everyone loves someone" as $\forall x. \exists y. L(x, y)$.

1. Define predicate functions as required.
   a. Define function $isZero(b, n)$ so that it returns $True$ if and only if the first n numbers in array b are all 0. You may assume that n is no larger than the size of the array b.
      - One attempt is to write the function as $isZero(b, n) \equiv b[0] = 0 \wedge b[1] = 0 \wedge b[2] = 0 \wedge \ldots \wedge b[n-1] = 0$. This is not correct, since "…" is not understandable. Instead, we should write:

$$isZero(b, n) \equiv \forall 0 \leq i < n. b[i] = 0$$

   b. Define function $isSorted(b, m, n)$ so that it returns $Ture$ if and only if subarray b between index m to n (including m and n) is sorted. You may assume that m and n are both valid index in array b and $n > m$. For example, $isSorted\big((1, 3, 5, 4), 0, 2\big) = True$.

$$isSorted(b, m, n) \equiv \forall i. m \leq i \leq n - 1 \rightarrow b[i] \leq b[i+1]$$

Types, Arrays, Expressions in Our Programming Language

- From here on, we will spend several lectures to clarify some definitions, notations, and grammar in our simple programming language.

(Data types)

- **Primitive data types** are: int (integers) and bool (Boolean).
- **Composite types**: (multi-dimensional) Arrays of primitive types of values, with integer indices.
  - Our purpose is to learn about program verification, so we keep the programming language as simple as possible.

(Expressions)

- **Expressions** in our programming language are "pieces of code" who have primitive type values. For example, you can consider a predicate as an expression with Boolean value. Expressions in our programming language can be built from:
  - **Constants**: Integers ($0, 1, -1$ …) and Boolean constants (T and F).
  - **Variables of primitive types**
  - **Functions** that return **primitive type values**

- **Operations:**
  - On integers: $+,\ -,\ *,\ /,\ \%,\ =,\ \neq,\ <,\ \leq,\ >,\ \geq,\ sqrt()$...
    - ❖ Note that, / and $sqrt()$ round toward 0 to an integer. For example, $13/\ 3 = 4$, $13\ /(-3) = -4$, and $sqrt(17) =\ 4$.
    - ❖ Division and mod by 0 and sqrt of negative values generate runtime errors.
  - On Booleans: $\neg,\ \wedge,\ \vee,\ \rightarrow,\ \leftrightarrow,\ =,\ \neq$ ...
  - On arrays: $size(\ )$ and array element selection. For example, $b = (0,3,4)$, then $size(b) = 3$.

- **Arrays:**
  - As usual, $b[e]$ is array element selection. Note that, e is an expression evaluates to an integer.
  - $size(b)$ gives the length of b.
  - In a multi-dimension array, $b[e_0][e_1] \ldots [e_{n-1}]$ is selecting the element with index e_0 in the first dimension, e_1 in the second dimension ... e_{n-1} in the n^{th} dimension. Note that, n is not a variable but an integer constant here. ("**...**" is not understandable). For example, $b = \big((6,3),(2,5,8)\big)$, then $b[1][2] = 8$.
  - Note that, we can never have an array appear by itself in an expression, it is always wrapped in some function, or we are selecting some element in the array.

- **Conditional:** $\textbf{if } B \textbf{ then } e_1 \textbf{else } e_2 \textbf{ fi}$
  - Semantically, if B evaluates to true, then evaluate e_1; if B evaluates to false, then evaluate e_2.
  - Note that, e_1 and e_2 are expression and we require them to have the same type.

- Note that:
  - We don't explicitly declare variables; we assume that we can infer the types. The default type is integer. For example: to have expression $p \vee x > 0$, we don't need something like "create variable x of type int".
  - An expression must evaluate to a primitive type of value, so it cannot evaluate to an array. For example: (assuming a and b are two arrays) $\textbf{if } B \textbf{ then } a \textbf{ else } b \textbf{ fi}[0]$ is illegal. But $\textbf{if } B \textbf{ then } a[0] \textbf{ else } b[0] \textbf{ fi}$ is legal.
  - Functions who return primitive type of values are allowed in an expression, but an expression cannot yield a function. For example: $\textbf{if } B \textbf{ then } f(x) \textbf{ else } g(x) \textbf{ fi}$ is legal; but $\textbf{if } B \textbf{ then } f \textbf{ else } g \textbf{ fi } (x)$ is not.

2. Are the following expressions legal?
   a. x/y — Yes
   b. $a[0{:}\ 2]$ — No
   c. $\textbf{if } x < 0 \textbf{ then } x * x \textbf{ else } sqrt(x) \textbf{ fi} + y$ — Yes
   d. $\textbf{if } i < 0 \textbf{ then } b[0] \textbf{ else } i \geq size(b) \textbf{ then } b[size(b) - 1] \textbf{ else } b[i] \textbf{ fi}$
      Yes, in our programing language, "$\textbf{if} - \textbf{else if} - \textbf{else}$" is "$\textbf{if } B_1 \textbf{ then } e_1 \textbf{ else } B_2 \textbf{ then } e_2 \textbf{ else } e_3$".

(Notations)

- Most commonly, c and d are constants; e and s are general expressions; B and C are Boolean expressions; a and b are array names; and u, v, etc. are variables. Greek letters like α and β stand for semantic values.

(Evaluate an expression)

- With a proper state, an expression can be evaluated to a value.

- For example: $\sigma = \{x = 5, y = 2\}$, then $\sigma(x * y) = 10$. Here, $x * y$ is an expression that we want to evaluate, and σ is a state that's proper for $x * y$.

- As an aside, here we have a question, when we evaluate an expression, how does something "syntactic" become something "semantic"? In other words, how is a programming language compiled into something meaningful to us? In this small section, to make this clear, I will use highlight to show values that are semantic.

3. Let $z \equiv 2 + 3$, evaluate $\sigma(z)$.
   - Since $z \equiv 2 + 3$, so we evaluate $\sigma(2 + 3)$, using the knowledge about "state as function" we discussed in lecture 3, we will evaluate $\sigma(2) + \sigma(3)$.
   - 2 and 3 are constants, we don't need any bindings in σ to find their values, thus $\sigma(2) + \sigma(3) = 2 + 3 = 2 + 3 = 5$

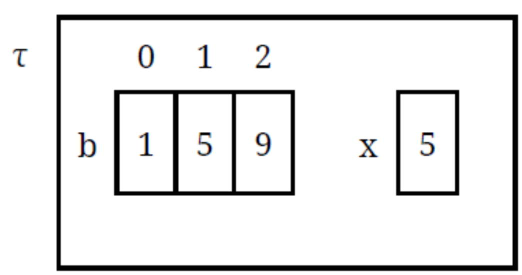
- In general, the value of $\sigma(e)$ depends on what kind of expression e is, so we use recursion on the structure of e (the base cases are variables and constants, and we recursively evaluate sub-expressions).
   - $\sigma(x)$ = the value that σ binds variable x to. For example, if $\sigma = \{x = 5\}$, then $\sigma(x) = 5$
   - $\sigma(c)$ = the value of the constant c. For example, $\sigma(5) = 5$. (σ is irrelevant here.)
   - $\sigma(e_1 + e_2)$ = the value of $\sigma(e_1)$ plus the value of $\sigma(e_2)$ [and similar for $-$, $*$, $/$ etc.].
   - $\sigma(e_1 < e_2) = T$ **iff** the value of $\sigma(e_1)$ is less than the value of $\sigma(e_2)$ [similar for $\leq$, $=$, etc].
   - $\sigma(e_1 \wedge e_2) = T$ **iff** the value of $\sigma(e_1)$ and the value of $\sigma(e_2)$ are both T [similar for $\vee$, $\rightarrow$ etc].
   - $\sigma(\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) = \sigma(e_1)$ if the value of $\sigma(B) = T$; it $= \sigma(e_2)$ if the value of $\sigma(B) = F$.

4. Let $\sigma = \{x = 1\}$, let $\tau = \sigma \cup \{y = 1\}$, and let $e \equiv (x = \textbf{if } y > 0 \textbf{ then } 17 \textbf{ else } y \textbf{ fi})$, evaluate $\tau(e)$.

$$
\begin{aligned}
\tau(x = \textbf{if } y > 0 \textbf{ then } 17 \textbf{ else } y \textbf{ fi}) &= \big(\tau(x) = \tau(\textbf{if } y > 0 \textbf{ then } 17 \textbf{ else } y \textbf{ fi})\big) \\
&= \big(1 = \tau(\textbf{if } 1 > 0 \textbf{ then } 17 \textbf{ else } y \textbf{ fi})\big) \\
&= (1 = \tau(\textbf{if } T \textbf{ then } 17 \textbf{ else } y \textbf{ fi})) \\
&= (1 = \tau(17)) \\
&= (1 = 17) \\
&= (1 = 17) = F
\end{aligned}
$$

(Arrays and their values)

5. How to write the following state τ in our language?



   - $\tau = \{b[0] = 1, b[1] = 5, b[2] = 9, x = 5\}$
     We take the **value of an array** to be a **function** from index values to stored values.

   - $\tau = \{b = \beta, x = 5\}$ where $\beta(0) = 1,\ \beta(1) = 5,\ \beta(2) = 9$
     If we give the function a name β (which is semantic), then we can write τ like this.

   - $\tau = \{b = \beta, x = 5\}$ where $\beta = \{(0, 1), (1, 5), (2, 9)\}$
     The function β can also be expressed as tuples (index, stored value).

- $\tau = \{b = \beta, x = 5\}$ where $\beta = (1, 5, 9)$
  The function β can also be simplified to a sequence of values.

- $\tau = \{b = (3, 5, 9), x = 5\}$

6. Let $\sigma = \{x = 1, b = \beta\}$ where $\beta = (2, 0, 4)$, evaluate $\sigma(b[x + 1] - 2)$.
   $$\begin{aligned}\sigma(b[x+1]-2) &= \sigma(b[x+1]) - \sigma(2)\\ &= \sigma(b)\big(\sigma(x+1)\big) - 2\\ &= \sigma(b)\big(\sigma(x) + \sigma(1)\big) - 2\\ &= \sigma(b)(1+1) - 2\\ &= \beta(2) - 2\\ &= 4 - 2 = 2\end{aligned}$$

<u>Updating a State</u>

- For any state σ , variable x , and value α , the "**update**" of σ at x with α , written $\sigma[x \mapsto \alpha]$, is the state that is a copy of σ except that it binds variable x to value α.
  - Note that, we are not really updating σ itself (although that is the traditional way to call this operation), that's why we quote the word "update": $\sigma\ [x \mapsto \alpha]$ is a new state and σ is not changed.
- We can give $\sigma[x \mapsto \alpha]$ a new name but we don't have to. We read $\sigma[x \mapsto \alpha](v)$ left-to-right — we're taking the function $\sigma[x \mapsto \alpha]$ and applying it to variable v.

7. Let $\sigma = \{x = 1,\ y = 2\}$, answer the following questions about state τ.
   a. Let $\tau = \sigma[x \mapsto 3]$, then $\tau = \{x = 3,\ y = 2\}$.
   b. Let $\tau = \sigma[z \mapsto 3]$, then $\tau = \{x = 1,\ y = 2,\ z = 3\}$.
      $\sigma(z)$ doesn't need to be defined (z is bind with a v
   c. Let $\tau = \sigma[x \mapsto 1]$, then $\tau = \{x = 1,\ y = 2\} = \sigma$.
      τ and σ are consist of the same bindings, they are not syntactically equivalent though (they are not the same state).