

Class 02: Propositional and Predicate Logic, Expressions, States, and Values

A. Why?

To verify programs, we'll use predicate logic to write out specifications and reason about them. You've more than likely already seen propositional and predicate logic in the past; it will be good to review the material and look at the notation we'll be using.

B. Outcomes

By the end of the class you should be able to

- Identify legal predicates when you see one
- Translate English descriptions of some simple predicates involving integers and arrays into (first-order) predicate logic.
- Write down and recognize definitions of memory states
- Understand what the textbook means by “the value of an expression (given some memory state.”

C. Propositional Logic

- Proposition letters (boolean variables), connectives \wedge , \vee , \rightarrow , \leftrightarrow , and \neg .
- Truth tables, logical equivalence, tautologies, contradictions
- Rules for transforming propositions: Commutativity of \wedge , \vee , \leftrightarrow ; Associativity of \wedge , \vee , \leftrightarrow ; Distributivity of \wedge over \vee and \vee over \wedge ; Transitivity of \rightarrow , \leftrightarrow ; Identity (T for \wedge , F for \vee); Domination (T for \vee , F for \wedge); Contradiction; Excluded middle; Idempotency of \wedge , \vee ($p \wedge p \leftrightarrow p \leftrightarrow p \vee p$); Double negation (Pierce's Law); DeMorgan's laws (\neg of \wedge ; \neg of \vee); Definition of \rightarrow using \neg and \vee ; Definition of \leftrightarrow using \rightarrow and \wedge

D. Predicate Logic (page 34)

- Add domains of discourse (e.g., integers, arrays of integers)
- Add primitive relations on domain values (e.g., $=$, \neq , \leq , etc).
- Add universal and existential quantification: $\forall x \in S : P$ and $\exists x \in S : P$ where S is a domain (set) of values and P is a predicate involving x .
 - E.g., every natural number > 1 is is $<$ its own square:
 - $\forall x \in \mathbb{N} : (x > 1 \rightarrow x < x^2)$
 - Leave off \in and S if they're understood.
- Two more DeMorgan's laws:
 - $(\neg \forall x : P)$ iff $(\exists x : \neg P)$
 - $(\neg \exists x : P)$ iff $(\forall x : \neg P)$

- Predicate functions
 - Often give names to predicates and parameterize them.
 - E.g. $\text{Even}(x) \equiv (x \% 2) = 0$ [where $\%$ is remainder operator]
 - E.g. $\text{SortedUp}(a, m, n) \equiv \forall i : m \leq i < n \rightarrow a[i] \leq a[i+1]$

E. Types and Expressions (page 24)

Start looking at programming language used in text.

- Expressions built from
 - Constants: Integer (0, 1, -1, ...) and Boolean constants (**true**, **false**).
 - “Simple” variables of types integer and Boolean (and maybe characters).
 - Array indexing expressions $a[s_1, s_2, \dots, s_n]$ (arrays yield simple values).
 - On integers: +, -, *, min, max, div, mod, =, \neq , <, \leq , >, \geq , divides
 - On booleans: \neg , \wedge , \vee , \rightarrow , \leftrightarrow , = (note = and \leftrightarrow mean the same)
 - Conditional expression: **if** B **then** s_1 **else** s_2 **fi** where s_1 and s_2 have the same simple type. ($B ? s_1 : s_2$) in C or Java etc.
 - Assign $x :=$ **if** $x \neq 0$ **then** y/x **else** -1 **fi**
- Notation: s for expressions in general, B for a boolean expression.
- No explicit declarations of variables; just assume we know what they are and what their types are. (E.g., x must be an integer in $x+2$.)
- Examples of expressions:
 - **if** B **then** $x+y$ **else** $x*y$ **fi**
 - **if** B **then** $a[i]$ **else** $a[\text{if } B' \text{ then } i \text{ else } j \text{ fi}]$ **fi** * **if** C **then** 5 **else** **if** C' **then** z **else** 17 **fi** **fi**
- Examples of non-expressions:
 - A (conditional) expression can't yield a function.
 - E.g. BAD: **if** $x > 1$ **then** min **else** max **fi**(t, u)
 - Don't have array-valued expressions.

F. States (page 29)

- The value of an expression or the truth of a predicate can depend on the values of the variables.
- A (memory) state (typically σ , τ) specifies values of variables.
 - E.g. $\sigma(x) = 1$ and $\sigma(y) = \text{false}$ but maybe $\sigma(z)$ is undefined.
 - $\sigma = \{ (x, 1), (y, \text{false}) \}$
 - For an array variable a , $\sigma(a)$ is a function from indexes to values.
 - E.g. $(\sigma(a))(0) = 5$, $(\sigma(a))(1) = 4$ for “in σ , $a[0]=5$ and $a[1]=4$.”
 - Write $\sigma(a)(0)$ for $(\sigma(a))(0)$ [fewer parentheses]
 - Or give $\sigma(a)$ a name: “Let α be a function with $\alpha(0)=5$, $\alpha(1)=4$, then $\sigma(a)=\alpha$.”

- $\sigma = \{ (a, \alpha) \}$ where $\alpha(0) = 5$ and $\alpha(1) = 4$
- Let $Var =$ set of all simple variables or array variables.
- A memory state maps a subset of Var to some values.
 - I.e., $\sigma: Var \rightarrow \mathcal{D}$ where \mathcal{D} is the domain of all values.
 - Technically if x is a variable of type T , then $\sigma(x)$ should $\in \mathcal{D}_T$, the domain of values of type T .

G. Semantics of Expressions (page 30)

- Textbook has two notations for “the value/meaning/semantics of expression s in memory state σ .”
 - They’ll usually use $\sigma(s)$ [extension of $\sigma(\text{variable})$ notation]
 - They start out using $\mathcal{S} \llbracket s \rrbracket (\sigma)$. (The \mathcal{S} is for “semantics.”)
 - The hollow brackets are hints that the argument is a syntactic object
- The notation is for precise communication.
 - ***It’s important to practice translating from notation to English!***
 - Other direction good but less critical.
- Definition of “the value of expression s in σ ” uses syntactic structure of s .
 - Base cases: s has no subexpressions
 - s is a constant or variable
 - Inductive cases: s does have subexpressions
 - E.g. unaryOp s_1 , s_1 binaryOp s_2 , $a[s_1]$, **if B then s_1 else s_2 fi** [has 3 subexpressions].
- Cases of definition:
 - If s is an integer or boolean constant c , then its meaning is itself.
 - $\sigma(c) = I(c) = c$.
 - $I(c)$ means “the standard interpretation of the constant c ”
 - If s is a simple variable x , its value is specified by the state.
 - $\sigma(x)$ [as a function on expressions] = $\sigma(x)$ [as a function on variables]
 - If s is s_1 op s_2 , then its meaning is the result of running the function denoted by the operator on the values of s_1 and s_2 .
 - $\sigma(s_1 \text{ op } s_2) = f(\sigma(s_1), \sigma(s_2))$ where f is $I(\text{op})$
 - $I(+)$ is addition on integers, $I(=)$ is equality on integers (or booleans),
 - (Unary operations like $-x$ are similar.)
 - If s is an array indexing expression $a[s_1, \dots, s_n]$ then its meaning is the result of running the function denoted by the array on the values of the indexes.
 - $\sigma(a[s_1, \dots, s_n]) = f(\alpha_1, \dots, \alpha_n)$ where $f = \sigma(a)$ and each $\alpha_i = \sigma(s_i)$.
 - Equivalent: $\sigma(a)(\sigma(s_1), \dots, \sigma(s_n))$.

- If s is a conditional expression **if** B **then** s_1 **else** s_2 **fi**, then its value is either the value of s_1 or the value of s_2 depending on whether the value of B is true or false.
 - If $s \equiv \mathbf{if } B \mathbf{ then } s_1 \mathbf{ else } s_2 \mathbf{ fi}$, then $\sigma(s) = \sigma(s_1)$ if $\sigma(B) = \mathbf{true}$ and $\sigma(s) = \sigma(s_2)$ if $\sigma(B) = \mathbf{false}$.