2 - values and types

value = anything that exist, that can be computed, stored, take part in data structure: constants, function return values...

↳ C types: int, char, long, pointer, array..

↳ Haskell types: bool, int, tuples, records, lists, functions...

type = set of values, equipped with one or more operations that can be applied uniformly to all these values ex: {true, false} → and, or, not

{...,-3,-2,0,1,2...} → add, multiply..

X {13, true, monday}

↳ no useful operations over it

$\boxed{\text{primitive types} =}$ values that cannot be decomposed into other sub values.

↳ C: int, float, double, char, long, short, pointers

↳ Haskell: bool, int, float, function values

↳ Python: bool, int, float, str, functions

every programming lang. provides built-in primitive types. some also allow programs to define new primitives

cardinality of a type = the number of distinct values. denoted as # Type

↳ # Bool = 2   # char = 256   #int = 2^{32}

• built-in types: can differ with the purpose of the lang. but some types are common but they have different names. ex: Integer - int

• user defined primitive types = enumerated types

↳ enum days {mon, tue, wed, thu, fri, sat, sun} → C++ values ⇒ days = {0,1

↳ ranges in Pascal and Ada

↳ type Population is range 0 ... 1e10;

(values ⇒ Population = {0, ..., 10^{10}})

type Month is (jan, feb, ... ,dec);

• discrete ordinal primitive types = datatypes values which have one to one mapping to a range of integers.

- they can be array indices, switch/case labels
- they can be used as for loop variables (in pascal)

↳ C: ordinal types

↳ Pascal, Ada: distinct types

$\boxed{\text{composite types} =}$ user defined types. composition of one or more other datatypes they are different types depending on the composition type.

↳ cartesian product (struct, tuples, records)

↳ disjoint union (C → union, pascal → variant record, haskell → data)
↳ mapping (arrays, functions)
↳ powerset (pascal → set datatype)
↳ recursive composition (lists, trees, complex data structures)

- cartesion product = values of several (mostly different) types are grouped into tuples

tuple ⇒ $S = \{a, b, c\}$, $T = \{1, 2\}$ } $S \times T = \{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\}$

$\#(S \times T) = \#S \cdot \#T$

multiple ∌ (a, b, c) $\{x, y, z\}$

homogeneout cartesion product ⇒ $S^n = S \times S \times S$ (whose components are both chosen from S

$S^0 = \{()\}$ → 0 tuple, not empty set (like void) in C

- disjoint union =

$S = \{1, 2, 3\}$, $T = \{3, 4\}$ } $S + T = \{$ left 1, left 2, left 3, right 3, right 4 $\}$

$\#(S+T) = \#S + \#T$

with these tags, we can check wheter the variant was chosen from S or T.
and we can all multiple of a value

- mappings = the set of all possible mapping

$S = \{a, b\}$, $T = \{1, 2, 3\}$ } $S \to T = \{ \{a \to 1, b \to 1\}, \{a \to 1, b \to 2\}, \{a \to 1, b \to 3\},$
$\{a \to 2, b \to 1\}, \{a \to 2, b \to 2\}, \{a \to 2, b \to 3\},$
$\{a \to 3, b \to 1\}, \{a \to 3, b \to 2\}, \{a \to 3, b \to 3\} \}$

$\#(S \to T) = \#T^{\#S}$

arrays	functions
only integer domain	all types of mappings possible
values stored in the memory	defined by algorithms
	efficiency, resource usage
	side effect, output, error, terminate problem

- powerset = the set of all subsets, restricted and special datatype, only in Pascal

$S = \{1, 2, 3\}$ $P(S) = \{ \{\emptyset\}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1, 2, 3\} \}$

$\# P(S) = 2^{\#S}$

↳ $\boxed{\text{recursive types} =}$ types including themselves in composition
in general they defined as: $R = \ldots + (\ldots R \ldots R \ldots)$

a recursive set equation always has at least solution that is a subset of every other solution

cardinality of a recursive type = infinite
↳ even if every individual value of the type is finite

ex: the set of lists is infinitely large, although every individual list in that set is finite

- lists = sequence of values $S = Int \times S + \{null\}$

  $List\ \alpha = \alpha \times (List\ \alpha) + \{empty\}$

- strings = sequence of characters

  classification
  - → primitive type, must be built-in = Python
  - → arrays of characters
  - → pointers to array of characters = C, C++
  - → lists of characters = Haskell, Prolog
  - → objects = Java

type systems

groups values into types

type check = checking the types of the operands before operation itself (always)

static typing

operands are checked at compile-time

↳ because each variables and expression has a fixed type

most of the high lev. lang

→ C/C++ (strict type checking)

→ Haskell, ML (type inference)

☆ More efficient, secure

dynamic typing

operands are checked after they are computed (executed), but before performing the operation, at run-time

↳ because values have fixed types, but variables and expressions donot have

→ python, prolog (interpreted lang)

☆ slower (bc of multiple check) and also uses extra storage

☆ no security

☆ more flexible

type equivalence

$T_1 \overset{?}{\equiv} T_2$ how to decide

name equivalence

types should be defined at the same exact place

→ most lang use

struct a { int int }

struct b { int int }

structural equivalence

types should have same value set (mathematical set equality)

☆ if their structures are same, even if their names are different you can compare them

☆ hard to implement

↳ it will give error if you compose or assign them.
(because their names are different)

$\boxed{\text{type completeness}}$

first order values

- assignment
- func. parameter
- take part in composition
- return value from a func

• functions → second order values in Pascal, Fortran
→ pointers in C
→ first order values in Haskell

C Types:

	Primitive	Array	Struct	Func.
Assignment	✓	×	✓	×
Function parameter	✓	×	✓	×
Function return	✓	×	✓	×
In compositions	✓	✓	✓	×

Haskell Types:

	Primitive	Array	Struct	Func.
Variable definition	✓	✓	✓	✓
Function parameter	✓	✓	✓	✓
Function return	✓	✓	✓	✓
In compositions	✓	✓	✓	✓

expressions

a segment that is evaluated, and produces a value

literals= retun itself 3 → 3
variable and constant access= a=3 → 3
aggregates to construct composite value without decleration/ definition x= (12,"ali")
variable references = pointers are not references
function calls
conditional expressions
iterative expressions = haskell