CA341 Data Types & Scope

Value - an entity which can be stored/manipulated by a program

**Types & operations**

Most languages group values into some type. Types are sets of values. If an expression is of type T then the result should be a value of type T

**Primitive Types**

Primitive values cannot be decomposed into smaller values

Primitive types have uniform operations

Boolean, Character, Integer, Float

Boolean - True, False

Character - ASCII characters, ISO Latin, UNICODE

Integer - 2\*32, signed and unsigned

Floating - Real Numbers

Cardinality of a type - number of distinct values in a type set

In C++ booleans are 1 or 0

In Java chars are integers

**Composite Types**

Composed of multiple simpler values

A data structure

Cartesian Products

Arrays

Disjoint Unions

Recursive Types - Lists, Trees

**Cartesian Product**

Values of several types grouped together

Pairs can be a construction of a pair of its values

Selection of first compent or second component

Cardinality is #(S x T) = #S x #T

Tuples that are from the same set are said to be homogeneous

For homogeneous sets: #(Sn) = #S x #S x (#S)n

If the cardinality n = 0, the empty tuple unit corresponds to the type void in C, Java and null record in Ada

**Mappings**

m:S → T

Represents mapping m from x in S to values y in T

If m maps the value x in S to the value y in T, we write y = m(x) then y is the image of x under m.

S → T represents the set of all mapping from S to T

Cardinality of Mapping #(S→ T) = (#T)#s

Since each value element S has #T values and there #S values in S

An array is an index sequence of elements with one elements of T for each index.

Finite Mapping, Lower bound/ Upper Bound

An construction of an array

Indexing elements

Functions are also mappings

Types can be mapped to other types.

**Disjoint Unions**

A vluae is selected from one or several possible different setes.

Consists of tags, values

Construction

Tag Test - determine if variant is from either set

Projection - Recover the original variant

**Recursive Types**

Defined in terms of itself

Lists

Homogeneous if all values are of same type, hetrogeneous

Construction

Length

Empty Test

Head

Tail

Concatenation

Strings

Sequence of Characters

Length

Equality

Lexicographical Comparison

Character

Substring

Concatenation

Some languages treat strings as types and must provide in built functions

Some languages treat strings as arrays of characters

**Type System**

Groups values into types

Prevents type errors from occurring

Statically typed language - Variable/Expression is fixed type, Checked at compile time

Dynamically typed language - Values typed, Variables are not, checked at run time

Static Typing

Efficient

Run-time checks of all operands invoked

Slows down execution

Values must be tagged to thier type

Dynamic Typing

Flexible and necessary where the data is not known before execution

**Type Equivalence**

Check if two types are equivalent.

Done with structural or name equivalence

Name equivalence is where two types are defined in the same place

Structural Equivalence is where two types made of same components

Expressions

Expressions evaluate to some value

Literals denote a fixed value of some type

Constructions are expressions that build composite value from its components

Function Calls

Operators

Conditional Expressions

If Expressions

Case Expressions

Iterative Expressions

Reference to a named constant is a constant access or variable access expression

**Bindings And Environment**

Binding is a fixed association between identifier and variable value or procedure

An environment or namespace is a set of bindings

Languages differ on binding entities

C Java Ada

Types values types

Variables local vars values

Function procedures instance vars variables

Class var procedures

Methods exceptions

Classes packages

Packages tasks

**Scope**

Declaration - Portion of programme text which declaration is in effect

Binding - Portion of programme text over which binding is applied

In early languages the declaration scope was the entire program. Each variable needs a unique name. Prevented in modern languages through use of blocks

**Blocks**

Delimiter for scope declarations

Monolithic - whole program

Flat Block - Several non-overlapping blocks

Nested Block - Blocks within blocks

Visibility

When an indentifier is bound to some entity X then there is a binding occurrence of that identifier

Applied occurrence entity

Block command contains local declaration and subcommand. Bindings produced by the block declaration are only in effect for the execution of the subcommand

Block expressions is an expression that contains local declaration and subexpression. Binding produced by the declaration are in effect of ealuation of subexpression.

**Static v Dynamic Scope**

Language is statically scoped if the body is executed in the environment od the procedure definition. Determine the binding occurrence by examining the source code

Dynamically scoped if the body of procedure is executed in environment of invocation. Cannot determine binding occurrences until run-time. Languages with dynamic scope cannot be statically typed.

**Declarations**

Type definition binds identifier to existing type

New Type declaration binds identifier to a new type that not equivalent to existing type.

Type declarations are new-type declarations

Constant declaration binds identifier to constant

Variable declaration create variable and binds identifier to the variable

Variable renaming binds an identifier to existing variable as an alias

Procedure definition binds an identifier to a procedure

Composing Declarations

Sequential Declarations composes sub declarations one after another

Collateral declarations composes sub declarations that are independent of each other

Recursive Declaration uses bindings that itself has produced

Not supported by older languages as they were inefficient on older hardware

Scope starts at beginning of declaration. Ends with enclosing block