

# The EXPRESS language

**A data modeling language defined by ISO  
TC184/SC4 and published as ISO 10303-11:1994**

# Data modelling

- **Data modelling is to capture the characteristics of a real world object or process using a formalised notation.**
- **Data modelling is important to be able to exchange information - not necessarily complete information, but adequate for a certain purpose (Abstraction).**
- **Non-IT examples of data modelling:**
  - Speech (meaning) is modelled in letters forming words and that together with a grammar defines a language used for the exchange of speech (meaning) using a non-audio medium. Information can be exact, such as the language used in legal contracts, or intentionally vague, such as language used in poetry.
  - Music is also modelled using a limited set of symbols.

# An abstraction of the real world

*Represents*

## *Real World*

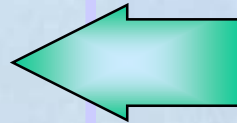
*Name of  
person*



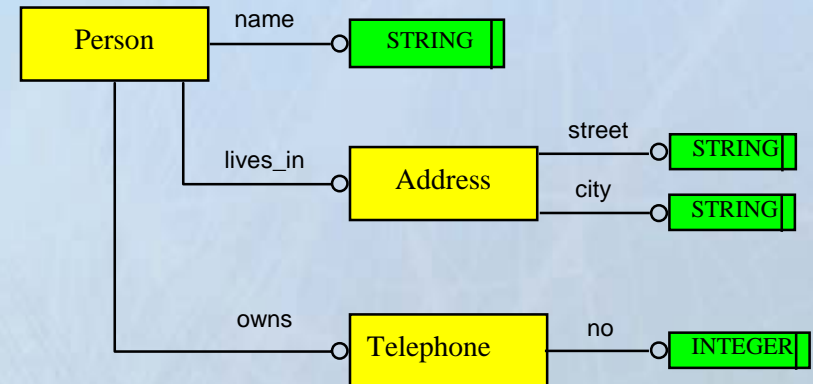
*Address*

- *Street*
- *City*

*Telephone  
number*



## *Abstraction*



```
ENTITY PERSON;
  name      : STRING;
  lives_in  : Address;
  owns      : Telephone;
END_ENTITY;
```

```
ENTITY Telephone;
  no : INTEGER;
END_ENTITY;
```

```
ENTITY Address;
  street : STRING;
  city   : STRING;
END_ENTITY;
```

# The EXPRESS language

- **EXPRESS is a data modeling language. It consists of language elements that allow an unambiguous data definition and the specification of constraints on those defined data.**
- **Published as ISO 10303-11 and used for most product data standards such as: ISO 10303 (STEP), ISO 13584 (PLIB), ISO 15331 (MANDATE), ISO 15926 (OIL&GAS), IFC, EDIF, IEC 6610, etc.**
- **Readable to humans and fully computer interpretable**
  - ➔ For example: a fax is computer readable, but needs human interpretation to process the information correctly.
- **EXPRESS is not a programming language, but ISO 10303 defines straight forward implementation forms.**
- **Textual and graphical notation.**
- **EXPRESS is case insensitive.**



# EXPRESS dialects

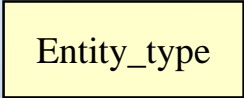

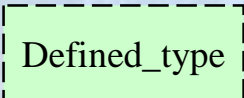
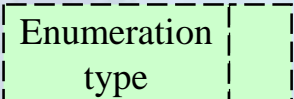
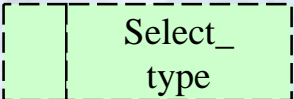
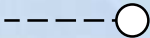
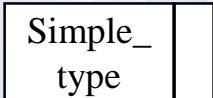
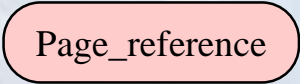
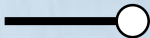
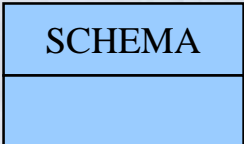
## ➤ **EXPRESS dialects within ISO 10303 :**

- EXPRESS (ISO 10303-11)- the textual, complete notation
- EXPRESS-G (ISO 10303-11) - the graphical, partial notation
- EXPRESS-I (ISO 10303-12) - instantiation language
- EXPRESS-X (ISO 10303-14) - mapping and view language

## ➤ **Proprietary dialects**

- EXPRESS-C
- EXPRESS +
- EXPRESS-V
- EXPRESS-M

# EXPRESS-G Overview

Symbols	Relationship lines	Text
	 Attribute	RT (redeclared attribute)
	SELECT	INV (INVERSE attribute)
	USE FROM	DER (DERIVE attribute)
	 Optional attribute	* (Constraint: Global RULE for ENTITY, WHERE rule or UNIQUENESS rule in ENTITY, WHERE rule in Type declaration)
	REFERENCE FROM	1 (ONEOF supertype constraint)
	 Subtype/supertype relation (inheritance)	& (AND supertype constraint)
		<b>Colour</b> Colour is not part of the EXPRESS-G specification, but has proved useful to increase the readability of the model

**Exercise:** review office schema,  
PDM schema

# EXPRESS identifiers

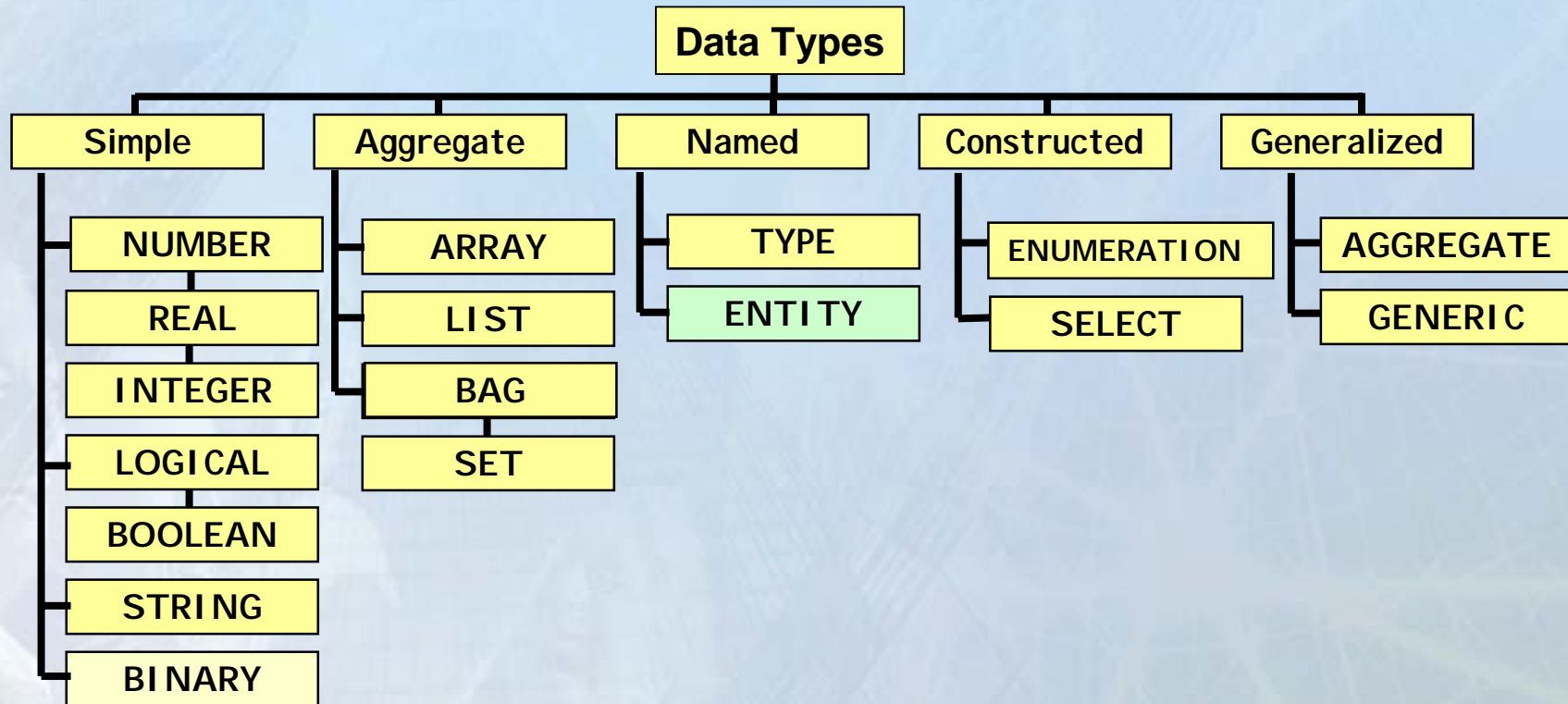
- **Identifiers are names given to the items declared in a schema, including the schema itself.**
  - The first character in an identifier shall be a letter. The remaining characters, if any, may be any combination of letters, digits and the underscore character
  - An identifier shall not be the same as an EXPRESS reserved word.
  - Identifiers are case insensitive.
  - No max length of identifiers

# EXPRESS reserved words

ABS	ELSE	FORMAT	NUMBER	SKIP
ABSTRACT	END	FROM	NVL	SQRT
ACOS	END_ALIAS	FUNCTION	ODD	STRING
AGGREGATE	END_CASE	GENERIC	OF	SUBTYPE
ALIAS	END_CONSTANT	HIBOUND	ONEOF	SUPERTYPE
AND	END_CONTEXT	HIINDEX	OPTIONAL	TAN
ANDOR	END_ENTITY	IF	OR	THEN
ARRAY	END_FUNCTION	IN	OTHERWISE	TO
AS	END_IF	INSERT	PI	TRUE
ASIN	END_LOCAL	INTEGER	PROCEDURE	TYPE
ATAN	END_MODEL	INVERSE	QUERY	TYPEOF
BACK	END_PROCEDURE	LENGTH	REAL	UNIQUE
BEGIN	END_REPEAT	LIKE	REFERENCE	UNKNOWN
BINARY	END_RULE	LIST	REMOVE	UNTIL
BLENGTH	END_SCHEMA	LOBOUND	REPEAT	USE
BOOLEAN	END_TYPE	LOCAL	RETURN	USEDIN
BY	ENTITY	LOG	ROLESOF	VALUE
CASE	ENUMERATION	LOG2	RULE	VALUE_IN
CONST_E	ESCAPE	LOG10	SCHEMA	VALUE_UNIQUE
CONSTANT	EXISTS	LOGICAL	SELECT	VAR
CONTEXT	EXP	LOINDEX	SELF	WHERE
COS	FALSE	MOD	SET	WHILE
DERIVE	FIXED	MODEL	SIN	XOR
DIV	FOR	NOT	SIZEOF	



# EXPRESS Data Types



# Simple data types I

- The simple data types define the atomic data units in EXPRESS and cannot be further subdivided into elements that EXPRESS recognize.

→ **NUMBER** - (ABSTRACT) all numeric values.

NUMBER	
--------	--

→ **REAL** - all rational, irrational and scientific real numbers

REAL	
------	--

— Precision (significant digits) can be specified REAL(5)

— 3.14      3.5e-5      1.E6

→ **INTEGER** - all integer number

INTEGER	
---------	--

— 1234567890    -9876543210

→ **LOGICAL** - **TRUE**, **UNKNOWN** or **FALSE**

LOGICAL	
---------	--

→ **BOOLEAN** - **TRUE** or **FALSE**

BOOLEAN	
---------	--

# Simple data types II

→ **STRING** - sequences of characters defined by ISO 10646

— '**single-quote string**' reads Single-quote string

— "**00000041**" reads A

— Max width-spec and FIXED can be used as constraints

— **STRING (10)**

— **STRING (2) FIXED**

→ **BINARY** - % symbol followed by one or more bits (0 or 1)

— **%110010101**

— Max width-spec and FIXED can be used as constraints

— **BINARY (10)**

— **BINARY (2) FIXED**

STRING	
--------	--

BINARY	
--------	--

# Aggregation (Collection) data types I

- **Aggregation data types have as their domains collection of values of a given base type - called: elements of the aggregate collection:**

- **ORDERED aggregates**

- ARRAY data type has as its domain indexed, fixed-size collection of like elements. It may use the optional UNIQUE to specify that an array cannot contain duplicate elements.

- ARRAY [1:10] OF INTEGER
    - ARRAY [-10:100] OF UNIQUE STRING(10)
    - ARRAY [1:100] OF OPTIONAL PERSON

- LIST data type has as its domain variable-size sequences of like elements. It may use the optional UNIQUE to specify that a list cannot contain duplicate elements.

- LIST [1:?] OF REAL
    - LIST OF UNIQUE PRODUCT



# Aggregation (Collection) data types II

## ➤ **UNORDERED aggregates:**

→ BAG data type has as its domain a variable-size, unordered collection of like elements in which duplication is allowed.

– BAG [1:100] OF NUMBER

– BAG OF ELEMENT

→ SET data type has as its domain a variable-size, unordered collection of like elements in which no two elements shall be instance equal

– SET [1:10] OF STRING (10) FIXED

– SET OF PERSON

## ➤ **Nested aggregates**

→ ARRAY [1:10] OF LIST OF DOCUMENT

→ LIST OF SET OF ARRAY[-10:10] OF INTEGER

# EXPRESS Data Types

➤ **Data types are provided as part of the EXPRESS language.**

➤ **They can be classified as:**

→ simple data types (NUMBER, REAL, INTEGER, STRING, BOOLEAN, LOGICAL, BINARY)

→ aggregation data types (ARRAY, BAG, LIST, SET)

→ named data types (TYPE, ENTITY)

→ constructed data types (SELECT, ENUMERATION)

→ generalized data types (AGGREGATE, GENERIC)

## Use of data types

Base  
data  
type

Param.  
data  
type

Underl  
data  
type



\*



➤ **Data types can also be classified according to their use in EXPRESS:**

- base data types - representation of an attribute, a constant or aggregation elements
- parameter data types - representation of a formal parameter, local variable or function result
- underlying data types - representation of a defined type

\*: only the defined data types, not entity data types

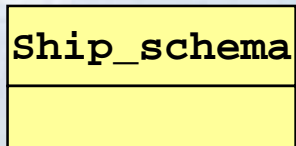
# EXPRESS language declarations

- **An EXPRESS declaration creates a new EXPRESS item associated with an identifier. This identifier can be referenced.**
- **The principle capabilities of EXPRESS are provided by the following global scope declarations:**
  - Data structure declarations
    - SCHEMA
    - ENTITY
    - TYPE
    - CONSTANT
  - Constraint declaration
    - RULE
  - Misc.
    - FUNCTION
    - PROCEDURE

# SCHEMA declaration

- A grouping/partitioning mechanism for an area of interest
- SCHEMA declarations defines common scope for a collection of entities and other data type declarations
- Declarations in one schema can be “used” in other schemata by means of interface declarations
- A SCHEMA has both a graphical and a lexical notation. At the SCHEMA level the notations are:

## EXPRESS-G



## EXPRESS

```
SCHEMA Ship_schema;  
END_SCHEMA;
```



# TYPE declaration

- **The TYPE declaration creates a new 'defined type' based on an 'underlying type'**
  - Used to distinguish conceptually different collections of values that happen to have similar representations
  - Used to increase the semantics of the underlying type
  - Increases the maintainability of the model
  - Local (WHERE) rules can be used to constrain the instantiation
- **Includes Constructed data types (SELECT and ENUMERATION)**

```
TYPE label = STRING; END_TYPE;
```

```
TYPE age = INTEGER;
```

```
WHERE SELF >= 0;
```

```
END_TYPE;
```



# TYPE declaration - SELECT

- **SELECT data type has as its domain the union of named data types in its select list**
- **It is used as a generalization of dissimilar types**
- **Typing may be necessary in population**

→ In case a component in a reference path is not an entity instance, and its type is not unambiguous from the declared domain, typing should/may be added. The typing should occur in front of the component:

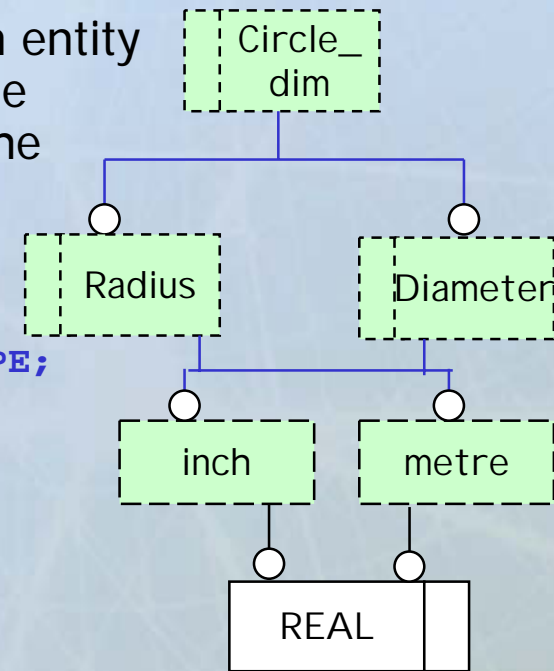
```
TYPE Circle_dim = SELECT (Radius, Diameter); END_TYPE;
```

```
TYPE Radius = SELECT (Inch, Metre); END_TYPE;
```

```
TYPE Diameter = SELECT (Inch, Metre); END_TYPE;
```

```
TYPE Inch = REAL; END_TYPE;
```

```
TYPE Metre = REAL; END_TYPE;
```



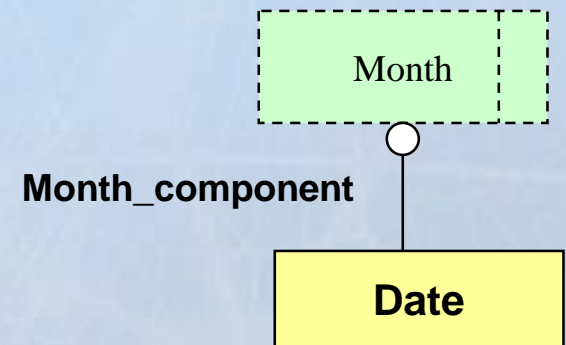
# TYPE declaration - ENUMERATION

## ➤ ENUMERATION data type that defines an ordered set of names - enumeration items:

- enumerations shall not have WHERE-rules (TC2);
- the ordering will be removed in edition 2.

```
TYPE month = ENUMERATION OF  
  (January, February, March, April,  
   May, June, July, August,  
   September, October, November,  
   December);  
END_TYPE;
```

```
ENTITY date;  
  Month_component : Month;  
END_ENTITY;
```



# ENTITY declaration

- ENTITY data type is a class - a representation of an entity establishing a domain of values defined by common **attributes** and **constraints** (local rules).

→ Sequence of optional declarations within an ENTITY:

- Supertype declaration (SUPERTYPE OF)
- Subtype declaration (SUBTYPE OF)
- Explicit attributes
- Derived attributes (DERIVE)
- Inverse attributes (INVERSE)
- Uniqueness rules (UNIQUE)
- Local rules (WHERE)

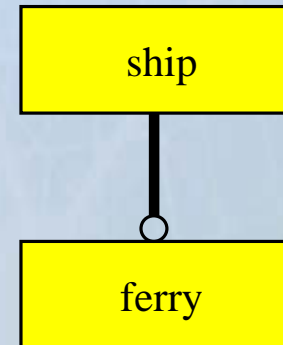
→ An ENTITY definition without any properties is valid:

```
ENTITY My_Entity;  
  
END_ENTITY;
```



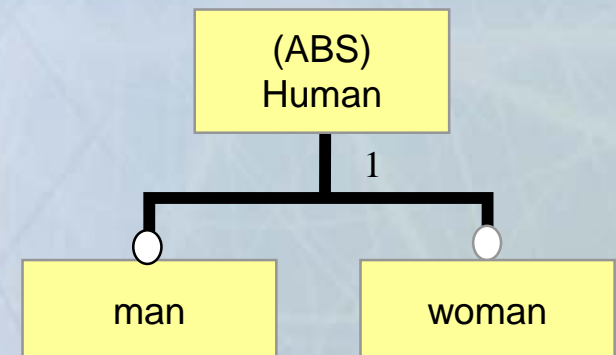
# Generalization/specialization

- **Subtypes and Supertypes are used to build a classification structure.**
  - The subtype is more specific than its supertype - supertypes are more general than their subtype(s)
  - A subtype inherits all the properties of its supertype; both attributes and constraints
  - Subtypes may have more than one supertype
  - Supertypes may have more than one subtype
  - A supertype may itself be a subtype
  - Subtype/supertype lattice is transitive
  - A subtype cannot be the supertype of itself, i.e. only acyclic graphs are valid

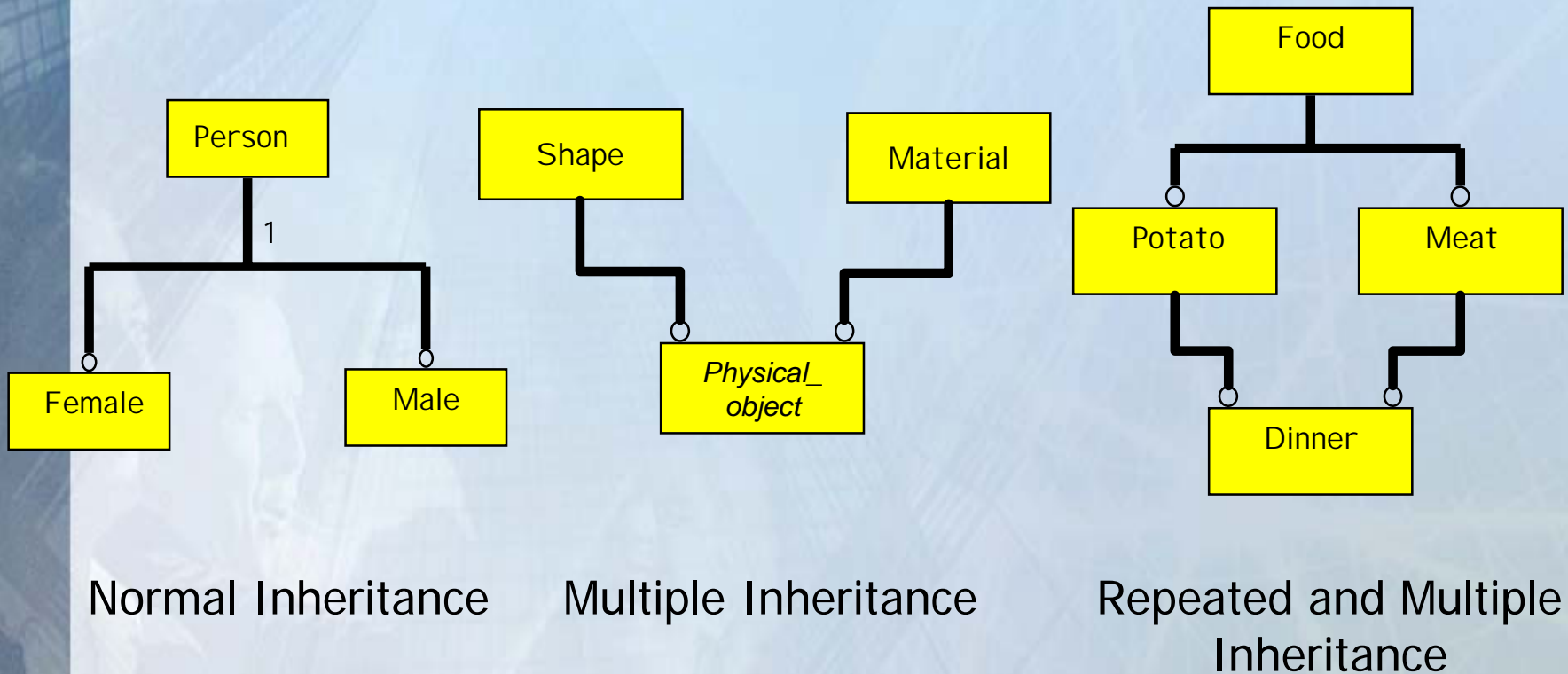


# Subtype/supertype

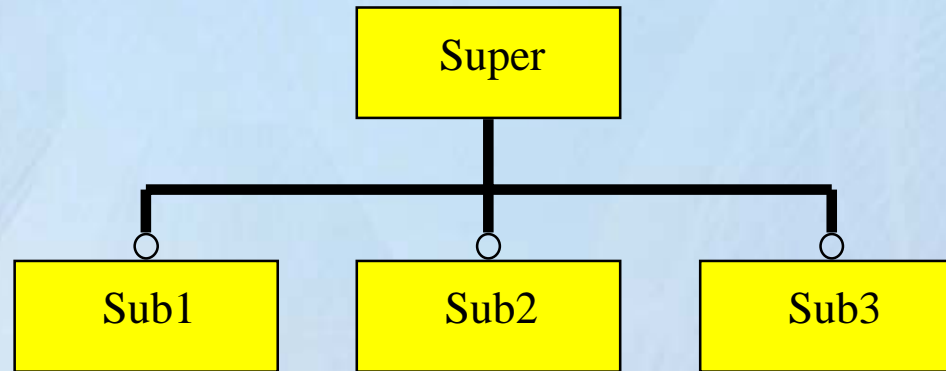
- **EXPRESS allows for inheritance**
  - The SUBTYPE OF defines the inheritance.
  - The (ABSTRACT) SUPERTYPE OF constrains this inheritance.
    - ABSTRACT means that the Supertype can only be instantiated as one of its valid specialization.
- **An entity (data type) instance which is a subtype, is an instance of each of its supertypes**
- **A complex entity data type establishes one particular combination of subtypes with a common supertype.**
  - The legal combination can be constrained by the SUPERTYPE OF declaration:
    - ONEOF     mutually exclusive
    - ANDOR    not mutually exclusive
    - AND



# Types of inheritance



# Subtype/Supertype constraints



➤ **ANDOR (default)**

**are not mutually exclusive, nor mutually inclusive 'all combinations'**

**sub1, sub1+sub2, sub1+sub2+sub3, sub1+sub3, sub2, sub2+sub3, sub3, super**

➤ **ONEOF**

**subtypes are mutually exclusive**

**sub1, sub2, sub3, super**

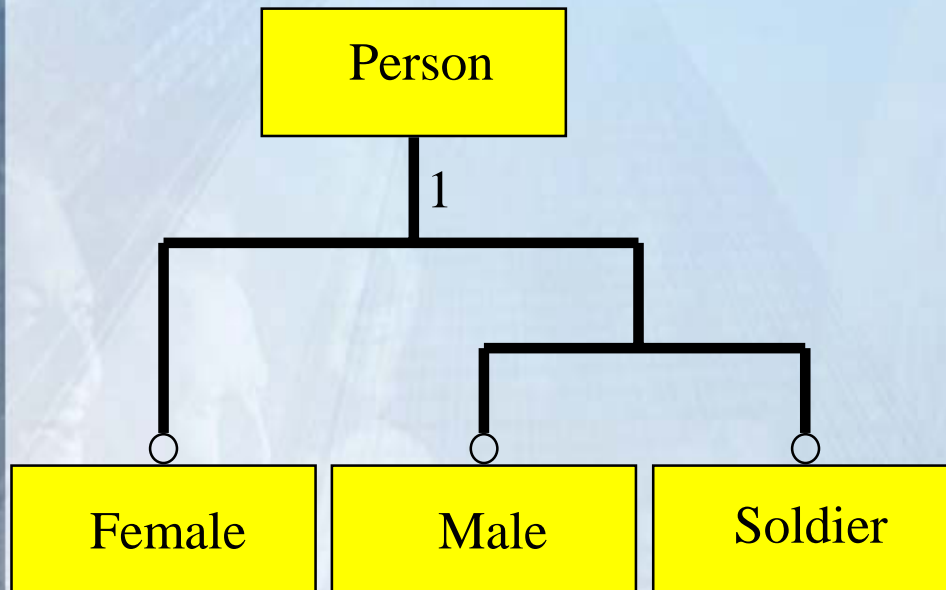
➤ **AND**

**subtypes are mutually inclusive**

**sub1+sub2+sub3, super**



# Example of Subtype/Supertype



Exercise: make airplane schema;  
document it

```
SCHEMA HUMANS;
```

```
ENTITY Person  
    SUPERTYPE OF (ONEOF  
        (Female, Male ANDOR Soldier));  
END_ENTITY;
```

```
ENTITY Female  
    SUBTYPE OF(Person);  
END_ENTITY;
```

```
ENTITY Male  
    SUBTYPE OF(Person);  
END_ENTITY;
```

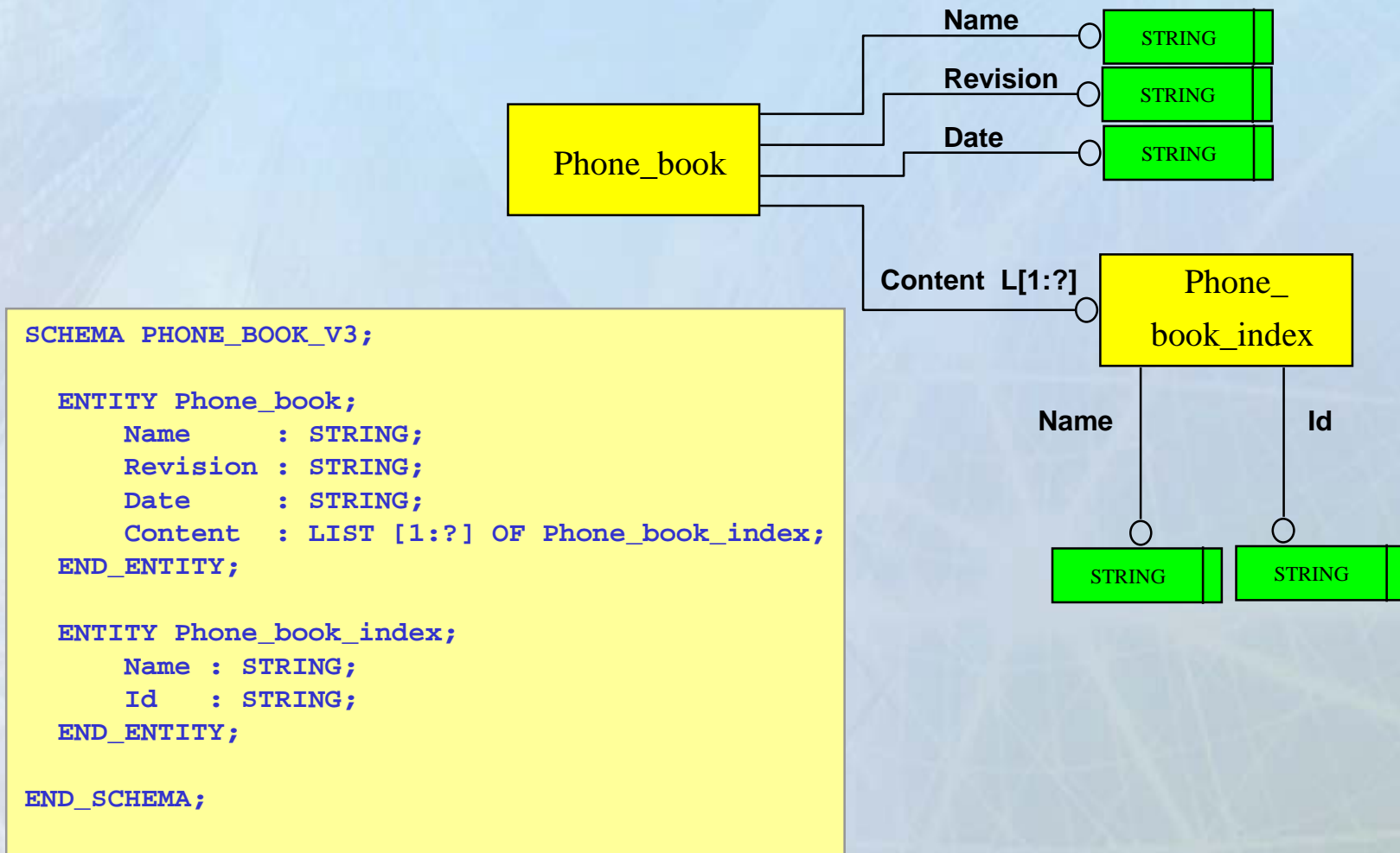
```
ENTITY Soldier  
    SUBTYPE OF(Person);  
END_ENTITY;
```

```
END_SCHEMA;
```

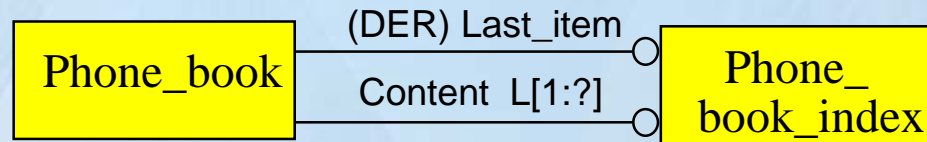
# ENTITY declaration - Attributes

- **Each attribute represents a property of the entity establishing a relationship between the entity data type and the data type referenced by the attribute.**
  - An explicit attribute represents a property whose value is used to create an instance. An explicit attribute can be declared as OPTIONAL.
  - A DERIVE attribute represents a property whose value is computed by evaluating an expression.
  - An INVERSE attribute defines a cardinality constraint for a relation in a particular role.

# Explicit attribute example



# Derived attribute example 1



```
SCHEMA PHONE_BOOK_V3;
```

```
ENTITY Phone_book;
```

```
    Content    : LIST [1:?] OF Phone_book_index;
```

```
    DERIVE
```

```
        Last_item : Phone_book_index := Content[HIINDEX(Content)];
```

```
END_ENTITY;
```

```
ENTITY Phone_book_index;
```

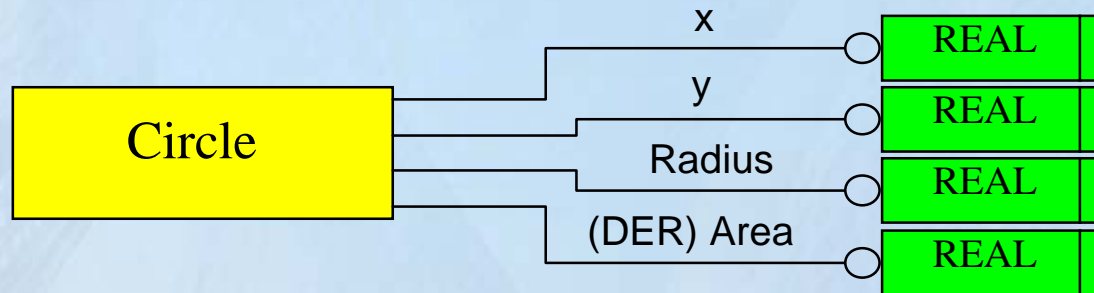
```
END_ENTITY;
```

```
END_SCHEMA;
```

Exercise: latest type of airplane



# Derived attribute example 2



```
SCHEMA Geometry;  
  
  ENTITY Circle;  
    x      : REAL;  
    y      : REAL;  
    Radius : REAL;  
    DERIVE  
      Area  : REAL := PI*Radius**2;  
  END_ENTITY;  
  
END_SCHEMA;
```

# Inverse attribute example

- The data type of the explicit attribute in the entity defining the direct relationship shall be one of the following:
- the current entity being declared,
  - a supertype of the current entity being declared,
  - a defined data type based on a select data type containing one of the above,
  - or an aggregation data type whose fundamental type is one of the above;

TC2

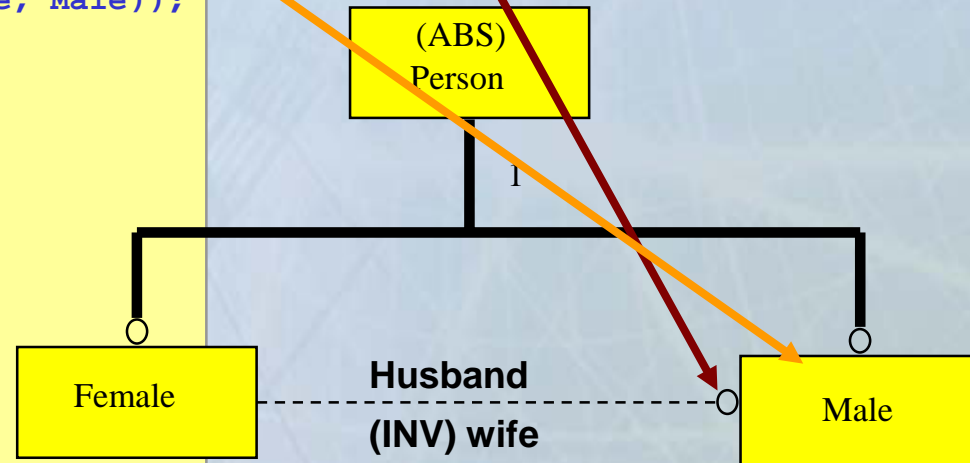
```
SCHEMA PHONE_BOOK_V1;
```

```
ENTITY Person
  ABSTRACT SUPERTYPE OF (ONEOF(Female, Male));
END_ENTITY;
```

```
ENTITY Female
  SUBTYPE OF(Person);
  Husband : OPTIONAL Male;
END_ENTITY;
```

```
ENTITY Male
  SUBTYPE OF(Person);
  INVERSE
    wife : Female FOR Husband;
END_ENTITY;
```

```
END_SCHEMA;
```



# Attribute redeclaration

- **An attribute declared in a supertype can be redeclared in its subtypes, according to the following principles:**
  - the domain of an inherited attribute can be specialised - not generalised; restricted - not expanded.
  - an explicit attribute can be changed to a DERIVE attribute.
  - an optional attribute can be changed to a mandatory attribute in a subtype - not vice versa.
  - the bound specification of LIST, BAG or SET may be constrained. The bound of an ARRAY can not be changed.

# Specialization

➤ **A specialization is a more constrained form of an original declaration. The following are defined specializations:**

- a subtype entity is a specialization of any of its supertypes
- a defined type is a specialization of the underlying type
- INTEGER and REAL are both specialization of NUMBER
- INTEGER is a specialization of REAL
- BOOLEAN is a specialization of LOGICAL
- LIST OF UNIQUE item is a specialization of LIST of item
- ARRAY OF UNIQUE item is a specialization of ARRAY of item
- ARRAY OF item is a specialization of ARRAY OF OPTIONAL item
- SET OF item is a specialization of BAG OF item



# Specialization cont.

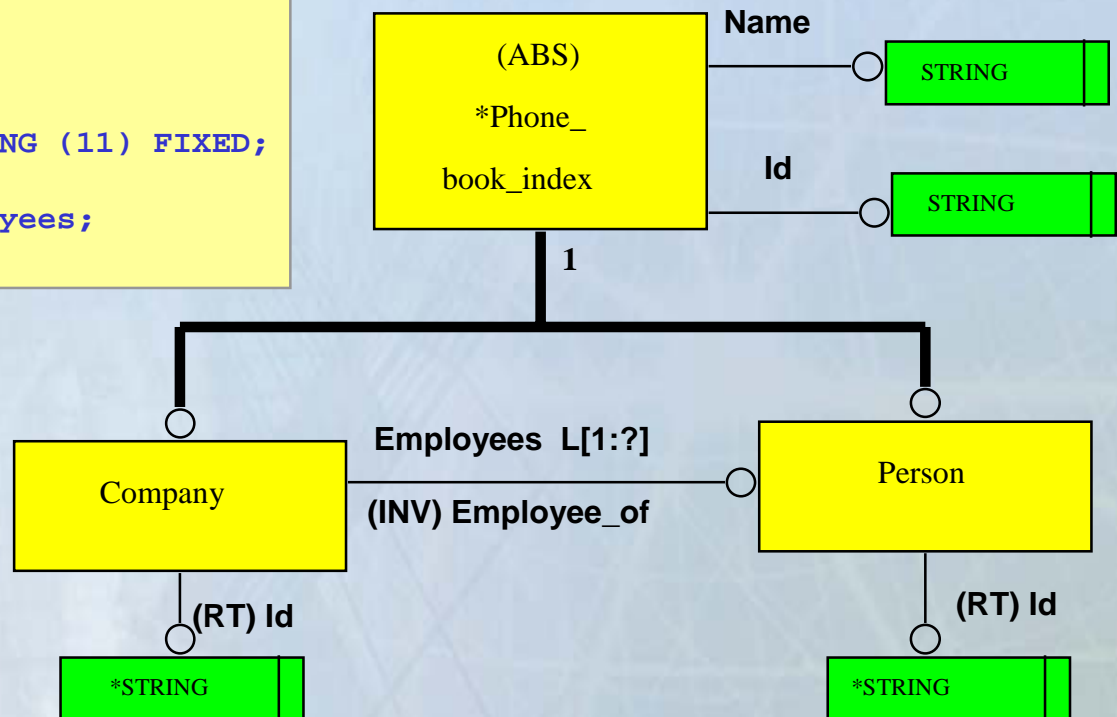
- letting AGG stand for one of ARRAY, BAG, LIST or SET then AGG OF item is a specialization of AGG OF original, provided that item is a specialization of original.
- letting AGG stand for one of BAG, LIST or SET then AGG[b:t] is a specialization of AGG[l:u] provided that  $b \leq t$  and  $l \leq b \leq u$  and  $l \leq t \leq u$
- letting BSR stand for one of the data types BINARY, STRING or REAL then
  - BSR(length) is a specialization of BSR
  - BSR(short) is a specialization of BSR(long) provided that  $\text{short} \leq \text{long}$
- A BINARY which uses the keyword FIXED is a specialization of variable length BINARY
- A STRING which uses the keyword FIXED is a specialization of variable length STRING

# Example of redeclared attributes

```
ENTITY Phone_book_index
  ABSTRACT SUPERTYPE OF (ONEOF(Company, Person));
  Name : STRING;
  Id : STRING;
END_ENTITY;

ENTITY Company
  SUBTYPE OF(Phone_book_index);
  Employees : LIST [1:?] OF Person;
  SELF\Phone_book_index.Id : STRING (14) FIXED;
END_ENTITY;

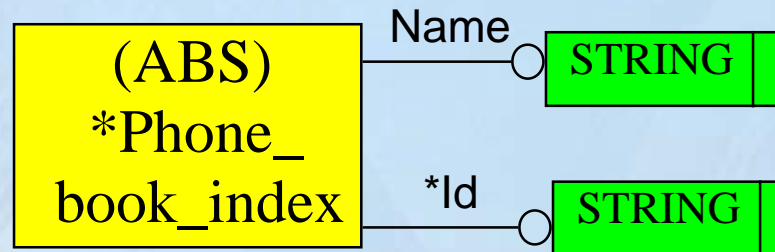
ENTITY Person
  SUBTYPE OF(Phone_book_index);
  SELF\Phone_book_index.Id : STRING (11) FIXED;
  INVERSE
    Employee_of : Company FOR Employees;
END_ENTITY;
```



# ENTITY declaration - Constraints

- **Local rules are assertions on the domain of entity instances and applies to all instances of that entity type**
  - Uniqueness rules defines a uniqueness constraint for individual or combinations of attribute values for all instances of this particular type in a population
  - Domain (WHERE) rules constrain the values of attributes for every entity instance
    - Each domain rule expression shall evaluate to either a logical (true, false, or unknown) value or indeterminate (?).
    - Every domain rule expression shall include a reference to SELF or attributes declared within the entity or any of its supertypes.
    - An occurrence of the keyword SELF shall refer to an instance of the entity being declared.
    - ...

# Example of an Uniqueness rule



```
ENTITY Phone_book_index
  ABSTRACT SUPERTYPE OF (ONEOF(Company, Person));
  Name : STRING;
  Id    : STRING;
  UNIQUE
    Unique_id : Id;
END_ENTITY;
```

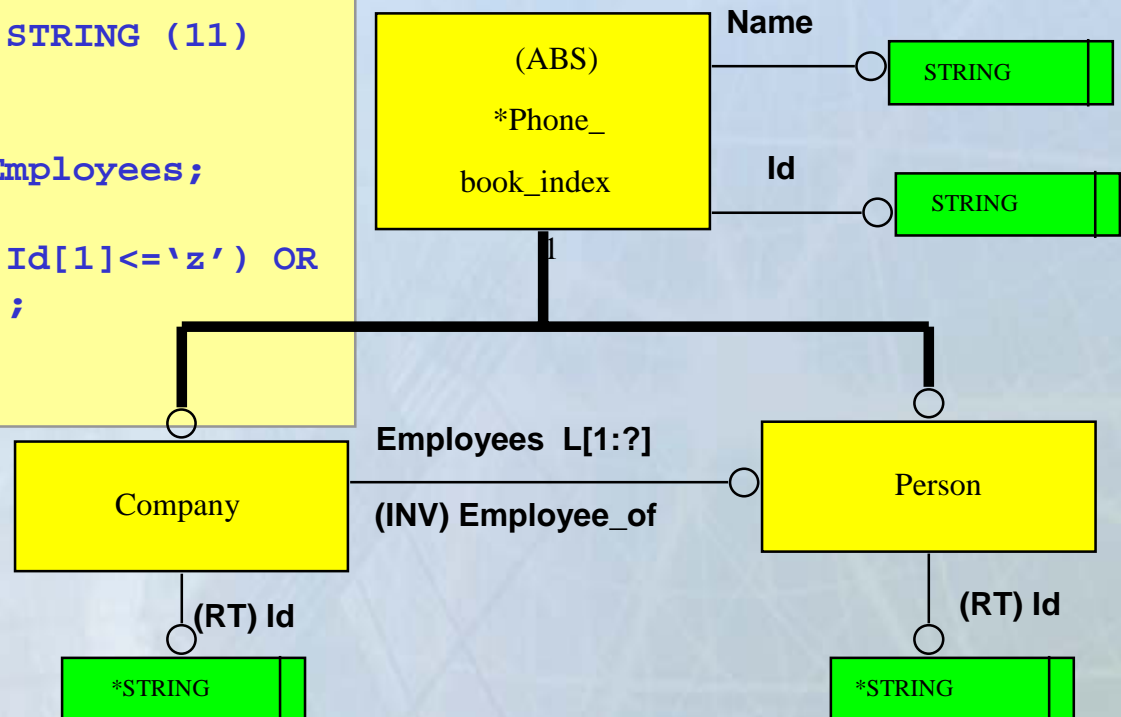


# Example of WHERE rule

```
ENTITY Phone_book_index
  ABSTRACT SUPERTYPE OF (ONEOF(Company,
  Person));
  Name : STRING;
  Id : STRING;
END_ENTITY;

ENTITY Person
  SUBTYPE OF(Phone_book_index);
  SELF\Phone_book_index.Id : STRING (11)
FIXED;
  INVERSE
    Employee_of : Company FOR Employees;
  WHERE
    Legal_id : (Id[1]>='a' AND Id[1]<='z') OR
    (Id[1]>='A' AND Id[1]<='Z');
END_ENTITY;
```

Exercise: add rule to airplane schema



# Implicit declaration - Entity constructor I

- **When an entity is declared, a constructor is implicitly declared. The constructor name is the same as the entity identifier, and the visibility of the constructor declaration is the same as that of the entity declaration.**
  - The constructor, when invoked, shall return a partial complex entity value for that entity data type to the point of invocation.
  - Each attribute value in this partial complex entity value is given by the actual parameter passed in the constructor call.
  - The constructor shall only provide the attributes which are explicit in a particular entity declaration.
- **When a subtype entity instance is constructed, the constructors for each of the components shall be combined using the || operator (complex entity construction operator, or entity concatenation operator)**

# Implicit declaration - Entity constructor II

```
ENTITY Point;  
    x,y,z  : REAL;  
END_ENTITY;
```

```
TYPE SEX = ENUMERATION OF (MALE, FEMALE); END_TYPE;
```

```
ENTITY Person;                ENTITY Adult Subtype of (Person);  
    Name : STRING;            Sex : Sex;  
    Age  : INTEGER;          END_ENTITY;  
END_ENTITY;
```

```
LOCAL  
    myPoint : POINT := Point(0.0, 0.0, 0.0);  
    myAdult : Adult := Person('John Smith', 50) || Adult (MALE);  
    ...  
END_LOCAL;
```

# CONSTANT declaration

- A Constant declaration is used to declare named constants. The scope of a Constant identifier shall be the function, procedure, rule, or schema in which the constant declaration occurs.

**CONSTANT**

**Thousand : INTEGER := 1000;**

**Million : INTEGER := Thousand\*\*2;**

**END\_CONSTANT;**

- **Built-in constants**

→ CONST\_E

→ Indeterminate (?)

→ PI

→ SELF

→ TRUE

→ FALSE

→ UNKNOWN



# FUNCTION & PROCEDURE declarations

- **A FUNCTION is an algorithm which operates on parameters and that produces a single result value of a specific data type.**
- **A PROCEDURE is an algorithm that receives parameters and operates on them to produce the desired end state.**
  - FUNCTION and PROCEDURE can have formal parameters specified by a name and parameter type.
  - A formal parameter to a PROCEDURE can be specified as VAR (variable) meaning that the actual parameter can be changed by the procedure.
    - The generalization data types (AGGREGATE and GENERIC) are used to allow a generalization of the data types that are used to represent the formal parameters.
    - Type labels can be used to relate the data types of type AGGREGATE and GENERIC.
  - LOCAL variables can be declared and used within a FUNCTION and a PROCEDURE and are not visible outside the actual FUNCTION or PROCEDURE declaration.
  - Local variables that are not initialized in the declaration are set to indeterminate upon each invocation of a FUNCTION and a PROCEDURE.

# FUNCTION and PROCEDURE declarations

```
FUNCTION bag_to_set(the_bag : BAG OF GENERIC : intype)
                        : SET OF GENERIC : intype;

    LOCAL
        the_set : SET OF GENERIC : intype := [];
    END_LOCAL;
    IF SIZEOF(the_bag) > 0 THEN
        REPEAT i := 1 TO HIINDEX(the_bag);
            the_set := the_set + the_bag[i];
        END_REPEAT;
    END_IF;
    RETURN (the_set);
END_FUNCTION;
```

Exercise: add function to airplane schema

```
PROCEDURE insert (VAR L : LIST OF GENERIC:GEN; E: GENERIC:GEN; P: INTEGER);
    LOCAL
        . . .
    END_LOCAL;
    (* Statements *)
END_PROCEDURE;
```

# RULE declaration

- Rules permit the definition of constraints that apply to one or more entity data types within the scope of a schema.
- A RULE declaration permits the definition of constraints that apply collectively to the entire domain of an entity data type, or to instances of more than one entity type.

```
ENTITY b;  
  a1 : c;  
  a2 : d;  
  a3 : e;  
UNIQUE  
  url : a1,a2;  
END_ENTITY;
```

```
RULE uniqueness_to_be_value_based FOR (b);  
(* further constrain the joint uniqueness in  
b to be value-based *)  
ENTITY temp;  
  a1 : c;  
  a2 : d;  
END_ENTITY  
LOCAL  
  s : SET OF temp;  
END_LOCAL  
REPEAT i := 1 TO SIZEOF(b);  
  s := s + temp(b[i].a1,b[i].a2)  
END_REPEAT;  
WHERE  
  wr1 : VALUE_UNIQUE(s);  
END_RULE;
```

# SCHEMA interfacing I

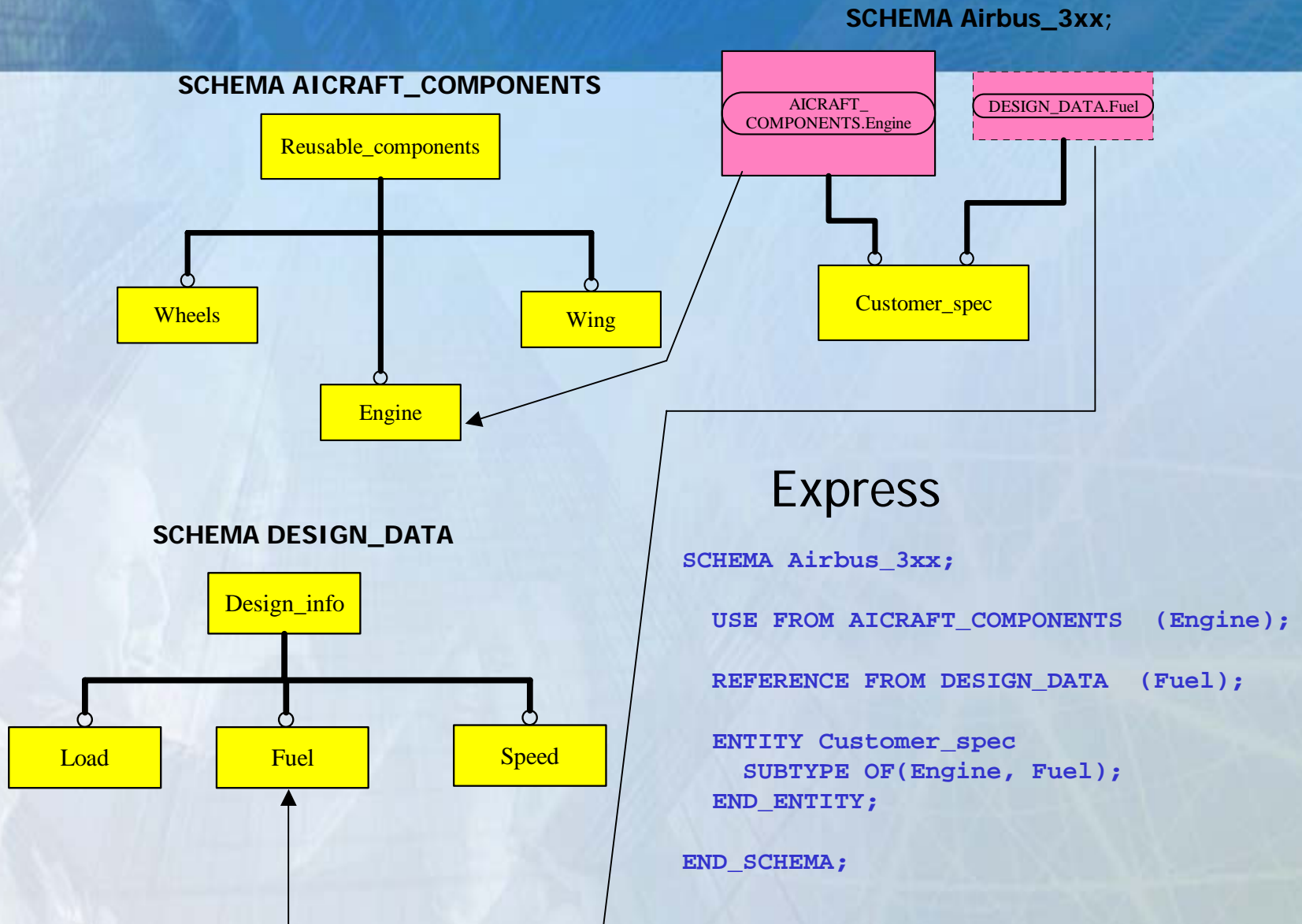
- **Interface specification is used to enable items declared in a 'foreign' schema to be visible in the 'current' schema.**
- **Used for modularization of a data model:**
  - building block concept; a collection of schemata constitutes one data model
  - reusable schemata; same schemata used in more data models
- **There are two interface declarations:**
  - USE FROM can interface ENTITY and TYPE declarations
    - declarations interfaced by USED FROM are handled as if locally declared
    - all ENTITY and TYPE declarations can be interfaced or only those specified in the USED FROM declaration
    - interfaced ENTITY and TYPE declarations can be renamed in the interfacing schema
    - chained interfacing; declarations indirectly interfaced through several schemata.



# SCHEMA interfacing II

- REFERENCE FROM can interface CONSTANT, ENTITY, TYPE, FUNCTION and PROCEDURE declarations
  - constraints in population of the entities that are interfaced by REFERENCE FROM
  - all CONSTANT, ENTITY, TYPE, FUNCTION and PROCEDURE declarations in the foreign schema can be interfaced, or only those declarations listed in the REFERENCE FROM declaration
  - interfaced declarations can be renamed in the interfacing schema
  - no chained interfacing

# Schema Interfacing III



# Short Form - Long Form schema

- **A 'short form schema' is a schema with the schema interface declarations intact.**
  - incomplete schema
  - can not be populated
- **A 'long form schema' is a schema without any schema interface declarations**
  - complete schema
  - can be populated
- **Needs EXPRESS tool to convert 'short form' schemata to 'long form' schema**

# Implicit interfaces I

**A foreign declaration may refer to identifiers which are not visible to the current schema. Those EXPRESS items referred to implicitly are required for a full understanding of the current schema.**

## ➤ **CONSTANT interfaces**

- ➔ any named data type, constant, function and procedure used in the declaration of that constant

## ➤ **TYPE interfaces**

- ➔ any defined types used in the declaration of the interfaced type, except for SELECT - none of the selectable items are implicitly interfaced.
- ➔ any constants, functions and procedures used in the declaration of the representation of the interfaced defined data type.
- ➔ Any constants, functions and procedures used within domain rules of the interfaced defined data type.
- ➔ any defined data types represented by a SELECT data type whose selection list contains the interfaced defined data type.



# Implicit interfaces II

## ➤ ENTITY interfaces

- all entity data types that are supertypes of the interfaced entity data type. Subtype/supertype graphs may be pruned by only following the SUBTYPE OF links when collecting the implicit interfaces of an interfaced entity data type.
- all rules referring to the interfaced entity data type and zero or more other entity data types, all of which are either explicitly or implicitly interfaced as a result of this interface.
- any constants, defined data types, entity data types, functions and procedures used in the declaration of attributes of the interfaced entity data type.
- any constants, defined data types, entity data types, functions and procedures used within the domain rules of the interfaced entity data type.
- any defined data types represented by SELECT data type which specifies the interfaced entity data type.

# Implicit interfaces III

## ➤ **FUNCTION interfaces**

- any defined data types or entity data types used in the declaration of parameters, returned type and local variables for the interfaced function.
- any constants, functions, and procedures used within the interfaced function

## ➤ **PROCEDURE interfaces**

- any defined data types or entity data types used in the declaration of parameters and local variables for the interfaced procedure.
- any constants, functions and procedures used within the interfaced procedure

## ➤ **RULE interfaces**

- any defined data types or entity data types used in the declaration of local variables within the interfaced rule.
- any constants, functions and procedures used within the interfaced rule.

**Exercise: interface another schema  
from the airplane schema**

# EXPRESSIONS

- **Expressions are combinations of operators, operands and function calls which are evaluated to produce a value.**
- **There are seven classes of operators**
  - Arithmetic operators accept NUMBER operands and produce NUMBER results
  - Relational operators accept various data types as operands and produce LOGICAL results
  - BINARY operators accept BINARY operands and produce BINARY result
  - LOGICAL operators accept LOGICAL operands and produce LOGICAL results
  - STRING operators accept STRING operands and produce STRING results
  - Aggregate operators combine aggregate values with other aggregate or individual elements and produce aggregate results
  - Component reference and index operators extract components from entity instances and aggregate values

# Operator precedence

Precedence	Description	Operators
1	Component references	[ ] . \
2	Unary operators	+ - NOT
3	Exponentiation	**
4	Multiplication/division	* / DIV MOD AND
5	Addition/Subtraction	- + OR XOR
6	Relational	= <> <= >= :: :<>: IN LIKE
<b>Note</b>    is the complex entity construction operator Modulo (MOD) and Integer division (DIV) produce an INTEGER result		



# Arithmetic operator

- Unary operators (+ -) are valid for NUMBER, REAL and INTEGER and require only one operand
- Exponential operator (\*\*) require two numeric operands
- Multiplication/Division (\* / DIV MOD) require two operands
- Addition Subtraction (+ -) require two operands

# Relational operators - Value Comparison

- **Value comparison operators are equal (=), not equal (<>), greater than (>), less than (<), greater than or equal (>=), less than or equal (<=) and are valid for:**
  - **Numeric comparison** - the value comparison operators when applied to numeric operands shall correspond to the mathematical ordering or real numbers.
  - **Logical comparison** shall observe the following ordering  

`FALSE < UNKNOWN < TRUE`
  - **Binary comparison** compare the bits in the same position in each value until an unequal pair is found or until all bit pairs have been examined.
  - **String comparison** compare the characters in the same position in each value until an unequal pair is found or until all character pairs have been examined.
  - **Enumeration element comparison** is based on their relative position

# Relational operators - Value Comparison

- **The value comparison (deep comparison) operators equal (=) and unequal (<>) are valid for:**
  - **Entity value comparison.** Two entity instances are equal by value comparison if their corresponding attributes are value equal
  - **Aggregate value comparison.**
    - Two aggregate values can only be compared if their data types are compatible.
    - Two aggregates are unequal if they contain an unequal number of elements
    - Aggregate elements are compared using value comparison operators.

# Relational operators - Instance comparison

- **Instance comparison(shallow comparison) operators are instance equal (:=:) and instance not equal (:<>:)**
- **The two operands of an instance comparison operator shall be data type compatible.**
- **For numeric, logical, string, binary, and enumeration instance comparison is equivalent to the corresponding value comparison.**
- **Aggregate instance comparison**
  - an aggregate comparison shall check the number of elements in each of the operands.
  - An aggregate comparison shall compare the elements of the aggregate value using instance comparison.
- **Entity instance comparison**
  - compare instance identifiers



# Relational & BINARY operators

## ➤ Membership operator `10 IN [0, 10, 100, 1000]`

- The membership operation (**IN**) tests an item for membership in some aggregate and evaluates to a LOGICAL value

## ➤ Interval expression `{5.0 < b <= 100.0}`

- An interval expression tests whether or not a value falls within a given interval. It contains three operands which types shall be compatible and be of a type having a defined ordering, i.e. simple types and defined types whose underlying types are either simple types or enumeration types.

## ➤ LIKE operator `a LIKE 'Adam Hall'`

- The Like operator (**LIKE**) compares two strings using pattern matching algorithms and evaluates to a logical value.

## ➤ Binary operators

- In addition to the relational operators, two further operations are defined:

- Binary indexing (`[]`) `b := a[5];`
- Binary Concatenation (`+`) `a := %101000101 + %101001`

# LOGICAL & STRING operators

## ➤ Logical operators

→ The logical operators are:

- **NOT** Requires one logical operand and produces a logical result
- **AND OR XOR**. Each need two logical operands and produces a logical result. The operators are commutative

## ➤ String operators

→ In addition to the relational operators, two further operations are defined:

- String indexing ( [ ] )
- String Concatenation ( + )

```
LOCAL
  name : STRING;
  c     : STRING;
END_LOCAL;
name := 'Eva'+ ' '+Hall'; -- concatenation
IF name[7:10] = 'Some' THEN
  c := name[1];
END_IF;
```

# Relational operators - Aggregate operators

- **The aggregate operators require two operands. The data type of the element selected is the base type of the aggregate value being indexed.**
  - Aggregate indexing (`[]`) accepts two operands: the aggregate value being indexed and the index specification, and evaluates to a single element from the aggregate value.
  - Intersection operator (`*`) accepts two aggregate value operands and evaluates to an aggregate value containing the elements that appear in both aggregates
  - Union operator (`+`) accepts two operands (one must be an aggregate) and evaluates to an aggregate value containing a union of the two operands.
  - Difference operator (`-`) accepts two operands (left-hand operand must be an aggregate) and evaluates to an aggregate value containing all elements in the left-hand operand except the values contained in the right-hand operand.
  - Subset operator (`<=`); **two aggregate operands; logical result**
  - Superset operator (`>=`); **two aggregate operands; logical result**



# Operators - References

- A simple reference is the name (identifier) of an item in a given scope. It is valid for attributes within an ENTITY declaration, Constants, ENUMERATION elements, ENTITY, FUNCTION, Local variables within the body of an algorithm, parameters within the body of an algorithm, PROCEDURE, RULE, Schemas within an interface specification, TYPE
- The prefixed reference is used when an ENUMERATION element is not unique.
- The attribute reference (.) provides a reference to a single attribute within an entity instance.
- The group reference (\) provides a reference to a partial complex entity value within a complex entity instance. A complex entity instance is an instance that has at least one supertype.



# Query operator

- Applies a logical expression against each element in an aggregate (source aggregate) and produce a new aggregate with all the elements in the source aggregate that the logical expression evaluates to TRUE for.

```
Procedure proc( ... );
```

```
Local
```

```
    persons      : set of person;
```

```
    male_adults  : bag of person;
```

```
End_Local;
```

```
    male_adults := QUERY(p <* persons | (p.sex = MALE)
                                AND
                                (p.age >= 18));
```

# Executable statements

- Executable statements are used to define logic and actions required to calculate value of DERIVE attributes and to validate constraints.
- Executable statements can be used in FUNCTION, PROCEDURE and RULE declarations.
- **EXPRESS statements:**
  - NULL statement

```
IF a = 13 THEN
    ; -- This is a NULL statement
ELSE
    b := a + 100;
END_IF;
```

# ALIAS Statement

- The ALIAS statement provides a local renaming capability for qualified variables and parameters.

```
ENTITY line;                                ENTITY point;
    start_point, end_point : POINT;        x, y, z : REAL;
END_ENTITY;                                END_ENTITY;

FUNCTION Calculate_length (the_line : LINE) : REAL;
    ALIAS s FOR the_line.start_point;
    ALIAS e FOR the_line.end_point;
    RETURN(SQRT((e.x - s.x)**2 + (e.y - s.y)**2 + (e.z - s.z)**2));
    END_ALIAS;
END_ALIAS;
END_FUNCTION;
```

# Assignment Statement

- **The Assignment statement is used to assign a value to a local variable or a parameter.**
  - the data type of the value must be assignment compatible with the data type of the actual local variable or parameter: the data type of the assigned value must be the same type or a specialisation of the type of the actual local variable or parameter.

```
PROCEDURE poo (VAR startp, VAR endp : POINT);  
LOCAL  
    a, b : REAL;  
    p : POINT;  
END_LOCAL;  
...  
a := 1.1;  
b := 2.5 * a;  
startp.x := p.x;  
...  
END_PROCEDURE;
```



# Compound Statement

- The Compound statement is a sequence of statements that can act as a single statement where this is required:

```
BEGIN  
    a := a + 1;  
    IF a > 100 THEN  
        a := 0;  
    END_IF;  
END;
```

# CASE Statement

- The CASE statement is a mechanism for selectively executing statements based on the value of an expression.

```
LOCAL
    a      : INTEGER;
    x, r    : REAL;
END_LOCAL;

...
a := 3;
x := 34.97;
CASE a OF
    1      : x := SIN (x);
    2      : x := EXP (x);
    3, 4   : x := SQRT(x);
    5      : BEGIN                -- start Compound statement
                        r := r + SQRT (x);
                        x := LOG (x);
                    END;          -- end Compound statement
    OTHERWISE: x := 0.0;
END_CASE;
```

# REPEAT Statement

- The REPEAT statement is used to conditionally repeat the execution of statements. Whether the repetition is started or continued is determined by evaluating the control condition(s). The control conditions are:

- finite iterations;
- while a condition is TRUE;
- until a condition is TRUE.

```
LOCAL
```

```
    i , n  : INTEGER;
```

```
    expr : LOGICAL;
```

```
END_LOCAL;
```

```
...
```

```
i := 50;
```

```
REPEAT i := 1 TO 100 BY 2  WHILE expr UNTIL n >= 1000;
```

```
    n := n + rand(i);
```

```
    -- I is 'repeat implicitly declared i'; not local variable i
```

```
...
```

```
END_REPEAT;
```

# REPEAT, SKIP, and ESCAPE Statements

- An ESCAPE statement can be used in a REPEAT statement to immediately terminate the REPEAT and transfer execution control to the first statement after the REPEAT statement.
- A SKIP statement can be used in a REPEAT statement to terminate the current iteration and transfer the execution control to the end of the REPEAT statement causing the UNTIL condition to be evaluated if defined.

```
REPEAT i := 1 TO 100;  
  
  ...  
  SKIP;  
  n := n + rand(i);  
  ESCAPE;  
  ...  
END_REPEAT; -- SKIP transfer execution control here, in REPEAT  
             -- ESCAPE transfer execution control to here, out of REPEAT
```



# Procedure call Statement

## ➤ The Procedure call statement invokes a procedure.

- The actual parameters provided with the call shall agree in number, order and type with the formal parameters defined in the declaration of the called procedure.
- The actual parameter passed shall be assignment compatible with the formal parameters;

...

```
INSERT (point_list, this_point, here);
```

...

```
PROCEDURE Insert (VAR l : LIST of GENERIC; e : GENERIC; p : INTEGER);
```

...

```
END_PROCEDURE;
```

# RETURN Statement

## ➤ The RETURN statements terminates the execution of a FUNCTION or PROCEDURE.

- The RETURN statement in a FUNCTION must specify an expression as argument. The data type of this expression must be assignment compatible with the attribute, local variable, parameter or constant that will receive the function value , i.e., the expression in the RETURN statement.
- The RETURN statement in a PROCEDURE shall have no arguments.

```
FUNCTION foo (p : person) : BOOLEAN;
```

```
...
```

```
    RETURN (TRUE);  -- function value as argument in RETURN statement
```

```
END_FUNCTION;
```

```
PROCEDURE proc (p: person; VAR parent, VAR child : PERSON);
```

```
...
```

```
    RETURN;  -- no return value
```

```
END_PROCEDURE;
```

# Arithmetic BUILT-IN-Functions

- **FUNCTION** **ABS** (V:NUMBER) : NUMBER;
- **FUNCTION** **ACOS** (V:NUMBER) : REAL;
- **FUNCTION** **ASIN** (V:NUMBER) : REAL;
- **FUNCTION** **ATAN** (V:NUMBER) : REAL;
- **FUNCTION** **COS** (V:NUMBER) : REAL;
- **FUNCTION** **EXP** (V:NUMBER) : REAL;
- **FUNCTION** **LOG** (V:NUMBER) : REAL;
- **FUNCTION** **LOG10** (V:NUMBER) : REAL;
- **FUNCTION** **LOG2** (V:NUMBER) : REAL;
- **FUNCTION** **SIN** (V:NUMBER) : REAL;
- **FUNCTION** **SQRT** (V:NUMBER) : REAL;
- **FUNCTION** **TAN** (V:NUMBER) : REAL;
- **FUNCTION** **VALUE** (V:STRING) : NUMBER;

# Other BUILT-IN-Functions I

- **FUNCTION** **BLENGTH** (V: BINARY) : INTEGER;  
→ returns number of bit in argument
- **FUNCTION** **EXISTS** (V: GENERIC) : BOOLEAN;  
→ returns FALSE if argument is indeterminate (?), else returns TRUE
- **FUNCTION** **FORMAT** (N: NUMBER; F: STRING) : STRING;  
→ formatting function
- **FUNCTION** **LENGTH** (V: STRING) : INTEGER  
→ returns number of characters in string argument
- **FUNCTION** **NVL** (V: GENERIC: GEN1;  
                    SUBSTITUTE: GENERIC: GEN1) : GENERIC: GEN1  
→ when first argument is indeterminate (?) then second argument is returned, else first argument is returned.
- **FUNCTION** **ODD** (V: INTEGER) : LOGICAL  
→ returns TRUE if argument is an odd number, else FALSE is returned



# Other BUILT-IN-Functions II

- **FUNCTION HiBound (V:AGGREGATE OF GENERIC) : INTEGER;**
  - returns the declared upper index when V is an ARRAY, else the declared upper bound of V is returned.
- **FUNCTION HiIndex (V:AGGREGATE OF GENERIC) : INTEGER;**
  - returns the declared upper index when V is an ARRAY, else the number of elements in V is returned.
- **FUNCTION LoBound (V:AGGREGATE OF GENERIC) : INTEGER;**
  - returns the declared lower index when V is an ARRAY, else the declared lower bound of V is returned
- **FUNCTION LoIndex (V:AGGREGATE OF GENERIC) : INTEGER;**
  - returns the lower index of the aggregate V.

# Other BUILT-IN-Functions III

## ➤ **FUNCTION ROLESOF (V:GENERIC) : SET OF STRING**

- returns a set of strings that specifies all roles played by the entity instance given as argument
- a role is a fully qualified attribute. A fully qualified attribute is defined to be the attribute name qualified with the schema name and the entity name, all in uppercase
- fully qualified attribute: 'MY\_SCHEMA.PERSON.LAST\_NAME'

## ➤ **FUNCTION SIZEOF (V:AGGREGATE OF GENERIC) : INTEGER**

- returns the number of elements in an aggregate

## ➤ **FUNCTION TYPEOF (V:GENERIC) : SET OF STRING**

- returns a set of strings that contains the fully qualified names of all the data type of which the argument is a member. Examples: 'MY\_SCHEMA.PERSON' , 'MY\_SCHEMA.MALE'
- simple types and aggregate types are returned without qualifications: 'INTEGER', 'STRING', 'ARRAY', 'LIST'

# Other BUILT-IN-Functions IV

- **FUNCTION USEDIN (T:GENERIC; R:STRING) : BAG OF GENERIC**
  - returns a bag of entity instances containing all entity instances that use the specified entity instance T in the specified role R.
  - When the give role R is empty or non existing, then the returned bag contains all entity instances that use the given entity instance T in any role.
  - Each entity instance will occur so many times in the returned bag as the number of occurrences of the given entity instance T in the specified role R.
- **FUNCTION VALUE\_IN (C:AGGREGATE OF GENERIC:GEN;  
V:GENERIC:GEN) : LOGICAL**
  - returns TRUE if the given argument V is a member in the given aggregate C. This function uses value comparison (deep compare)
- **FUNCTION VALUE\_UNIQUE (C:AGGREGATE OF GENERIC) : LOGICAL**
  - returns TRUE when all elements in the given aggregate C are value unique, else FALSE or UNKNOWN are returned

# BUILT-IN-Procedures

➤ **PROCEDURE INSERT (VAR L : LIST OF GENERIC:GEN;  
E: GENERIC:GEN;  
P: INTEGER);**

→ inserts the element E in position P in the list L.

➤ **PROCEDURE REMOVE (VAR L : LIST OF GENERIC; P INTEGER);**

→ removes element in position P from the list L

Exercise: finally - a global rule