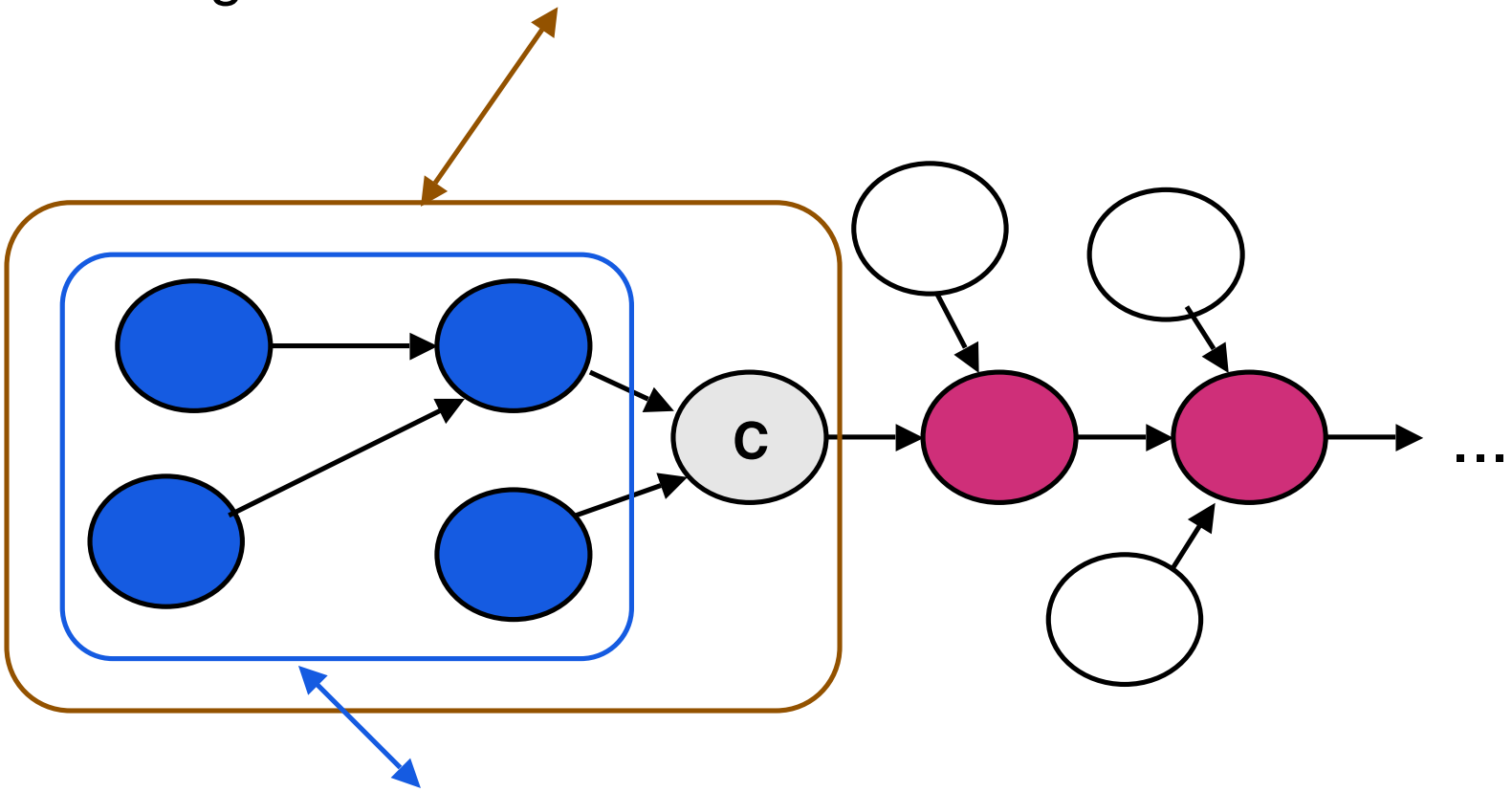
Inheritance
Polymorphism & Dynamic Types

Inheritance Terminology

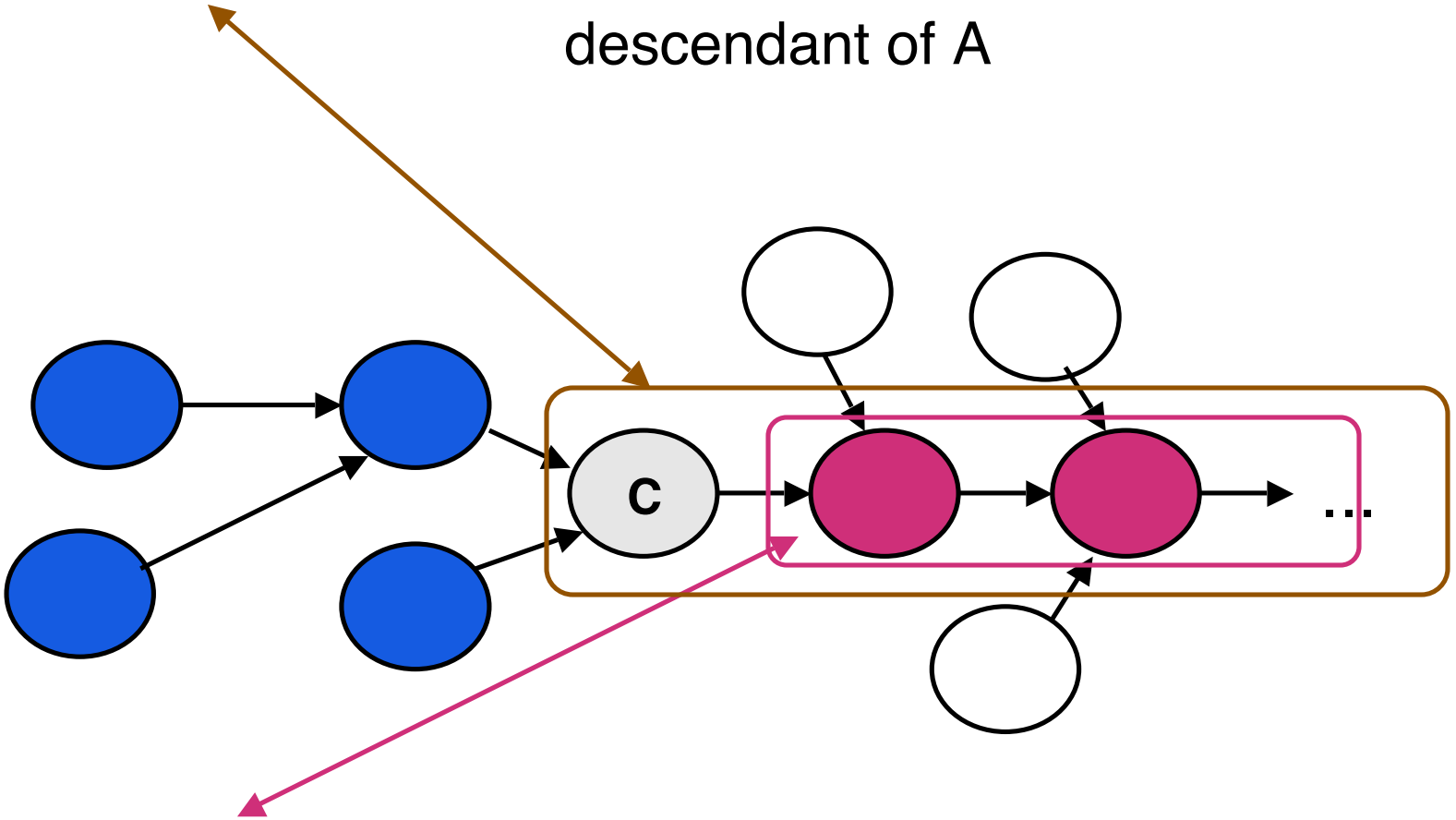
- Any class that inherits directly or indirectly from C, including C itself is **descendant** of C



- A **proper descendant** of C is a descendant of C other than itself

Inheritance Terminology – 2

- An **ancestor** of C is a class A such that C is descendant of A

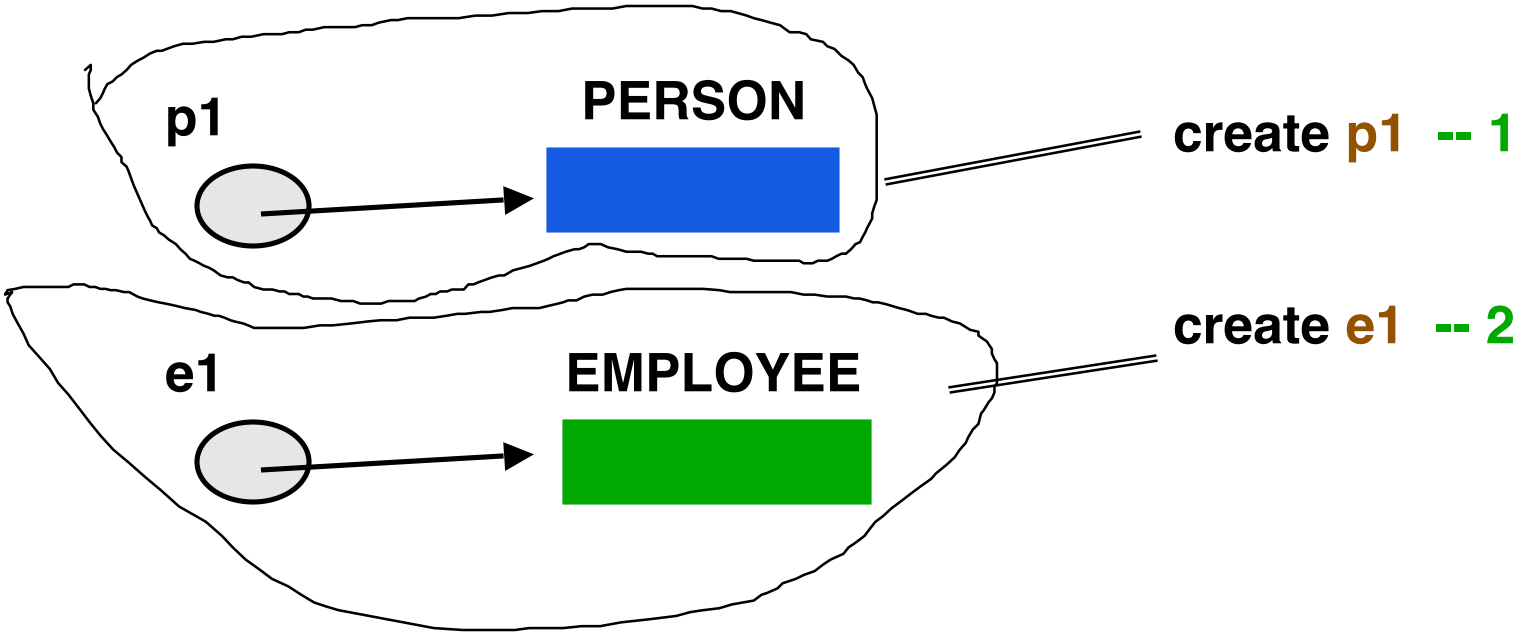


- A **proper ancestor** of C is an ancestor of C other than itself.

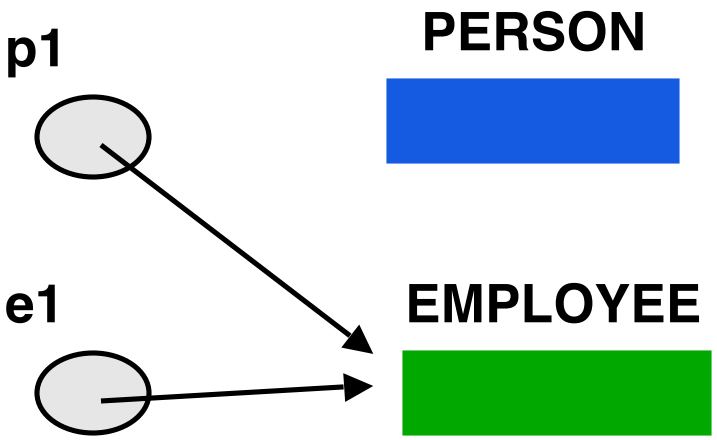
Subtyping Inheritance

- Subtyping relationship
  - » **Occurs when there is a strong degree of commonality between two or more classes**
    - > **E.g. between PERSON and EMPLOYEE**
- An EMPLOYEE is a PERSON
  - » **employees behave like persons but also have their own specialized behaviour**
- When this degree of common behaviour occurs, EMPLOYEE is said to be a **subtype** of PERSON
- Subtyping models the **is-a** relationship between classes

Dynamic Binding



p1 := e1 -- 3



Dynamic Binding – 2

- It is the essence of **polymorphism** – multiple types
  - » **Ability to invoke methods applicable to the dynamic type of an object rather than its static type**
    - > **During execution we can attach a reference to objects of different types**
    - > **both PERSON and EMPLOYEE have a feature display (EMPLOYEE inherits from PERSON)**

```
p1 , p2 : PERSON
e : EMPLOYEE
p2 := p1        -- ok type match
p1.display      -- PERSON display
p1 := e         -- ok, type conforms
p1.display      -- EMPLOYEE display
```

Example hierarchy

- Consider the following class hierarchy

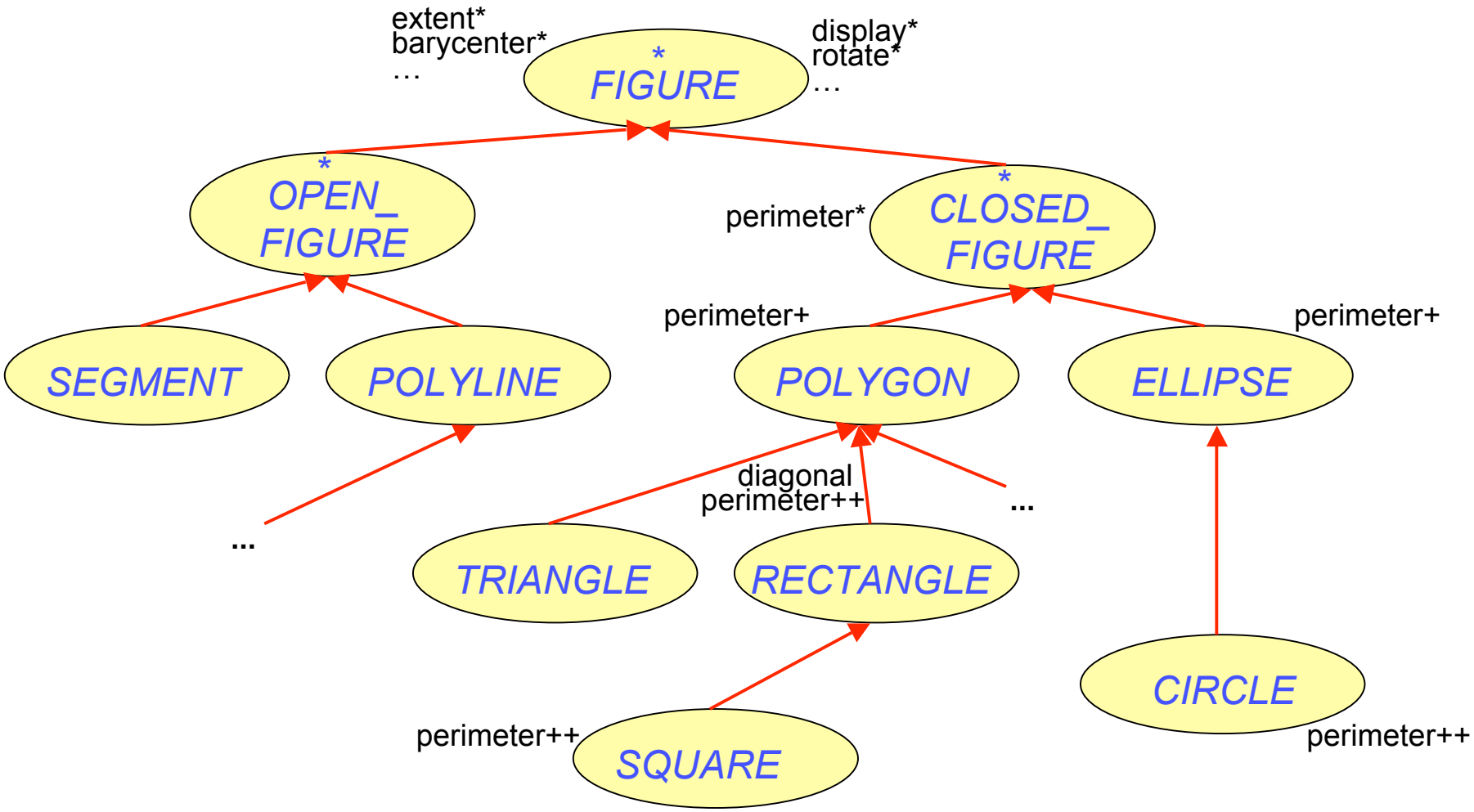


Figure Polymorphism

- Consider a figure hierarchy similar to that on page 468
- Suppose we had an array of figures and want to rotate all the figures in an array of figures

  > **Each figure has its own rotation method**

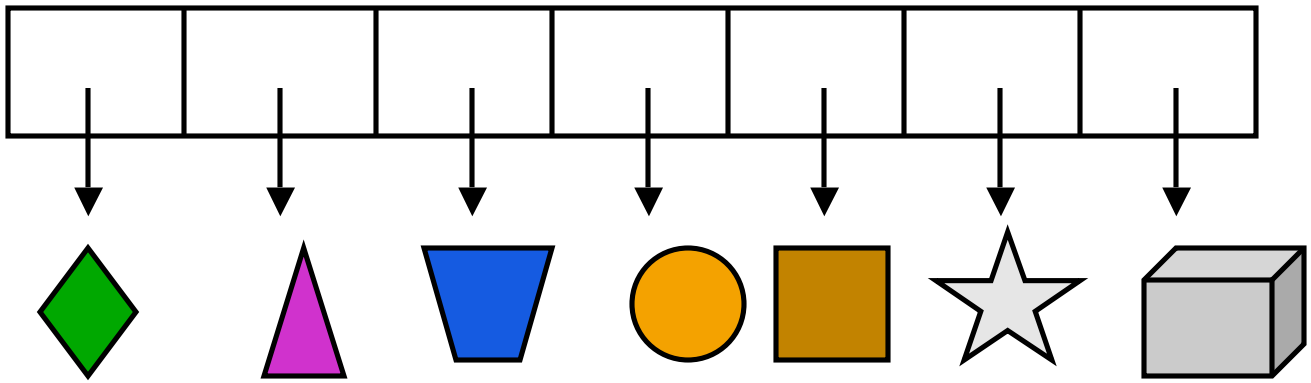


Figure Polymorphism – 2

- Want a general and maintainable solution
- Want to be able to add new kinds of figures without
  - » **breaking previous programs**
  - » **without modifying the rotate all figures method**
- Solution
  - » **dynamic binding**

Figure Polymorphism – 3

```
-- In a parent class

f : ARRAY [ FIGURE ]

rotate_all ( d : real ) is
    require d > 0
    do
        from  i := 1
        until i > f.upper
        loop
            f.item(i).rotate(d)  -- dynamic binding
            i := i + 1
        end
    end -- rotate_all
```

Feature Call Rule

- In a feature call **x.f** where the type of **x** is based on a class **C**, feature **f** must be defined in one of the ancestors of **C**
  - » **Example in rotate_all**
    - > **rotate must be a feature in the class Figure**
    - > **Each type of figure creates a custom instance of the feature rotate**

Type Conformance Definition

» A type U conforms to a type T only if the declared class of U is a descendant of the declared class T

» For generically derived types, every actual parameter of U must (recursively) conform to the corresponding formal parameter in T

> void does not conform to expanded types

Type Conformance Rule

An attachment of target **x** and source **y** is only valid if the type of **y** conforms to the type of **x**

Attachment is either

x := y

or

y is an actual argument to parameter x

Direct Instances & Instances

- A direct instance of a class C is an object produced according to the exact definition of C , either

  through a creation instruction, create x , where the target x is of type C

  or

  recursively by cloning a direct instance of C

- An instance of C is a direct instance of a descendant of C

Static & Dynamic Types

- Static-dynamic type consistency
  - » **An entity declared of type T may, at run time only, become attached to instances of T**
- Static type is the type of the variable declared in the program text
- Dynamic type is the type of the instance attached at execution time
- The type of **void** is **NONE**

Assignment Attempt

- Type rules ensure statically verifiable dynamic behaviour
  - » **No surprises at run time**
- But type rules are too restrictive, consider

  figlist : LIST [FIGURE]
  - » **What is the max diagonal of rectangles in the list?**

    figure := rectangle ; figure.diagonal **Wrong**
  - » **Cannot solve as diagonal is not a feature of FIGURE**
    - > **Do not want to have diagonal as a part of FIGURE as all figures would need to define it**
  - » **Another example, remove all circles from the list**

Assignment Attempt – 2

- We need to be able, in some circumstances, to know the dynamic type of an object
- Assignment attempt makes the assignment if the dynamic and static types conform, otherwise it returns void
  - » **subtype_object ?= supertype_object**

Assignment Attempt Example

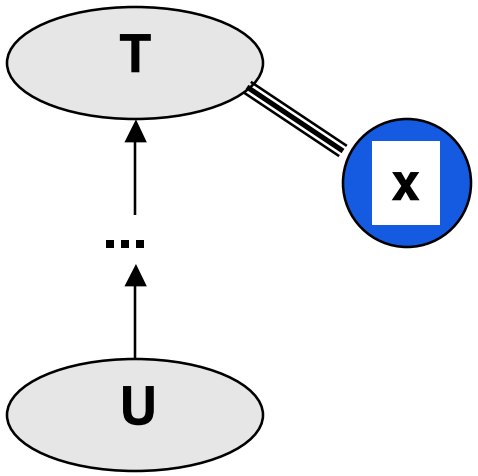
```
maxdiag ( figlist : LIST [ FIGURE ] ) : REAL is
    require  list_exists: figlist /= Void
    local  r : RECTANGLE
    do
        from figlist.start ; Result := 0.0
        until  figlist.after
        loop
            r ?= figlist.item
            if r /= Void then
                Result := Result.max(r.diagonal)
            end
            figlist.forth
      end
    end
```

Attempt assignment

Check if successful

Polymorphic Creation

- Assume **x** is of static type **T** but we want to assign to **x** an instance of static type **U** where **U** is a descendant of **T**



Use

create {U} x.make(…)

Obsolescent use of !!
! U ! x . make (...)