

PNYP: Eiffel jegyzet

Contents

Gyakorlati jegyzet	2
Általános információk, elvek	2
Vizsgán ezekre oda kell figyelni	2
Szintaxis, konvenciók	2
Láthatóság	3
Alapvető műveletek, operátorok, konstansok	3
Void-biztonság, attached , detachable	3
Osztályok	4
Rutinok: eljárások és függvények	4
Operátorok	5
Attribútumok (adattagok)	5
Példányok létrehozása, creation procedure	5
default_create	6
Szerződések	6
old operátor	7
Frame problem, frame rule, frame condition	7
Ellenőrzések do klózban	7
Felüldefiniált rutin elő- és utófeltétele	7
Hoare-hármas	7
Utasítások	8
Elágazás	8
Sokágú elágazás: inspect	8
Általános ciklus	8
Bejáró ciklus	9
Kvantálás	9
Öröklődés	9
Rutinok felüldefiniálása	10
Kifejtett osztályok öröklődése	10
Privát öröklődés	10
Örökölt rutinok láthatóságának megváltoztatása	10
Absztrakt osztályok, rutinok	10
Többszörös öröklődés	11
Névütközések feloldása	11
Dinamikus kötés, late binding, dynamic dispatch	11
Ismételt öröklődés	13
Variancia	13
Liskov helyettesítési elvet nem sértő variancia	13
Kovariáns paraméter példa	13
CAT: Changed Availability or Type	14
Generikus osztály	14
Tuple	14
Egyenlőségvizsgálat és másolás	15
Referencia típus egyenlőségvizsgálata	15
Kifejtett típus egyenlőségvizsgálata	15

Konverzió	15
Kivételkezelés	16
Kivételkezelés fajtái	16
<code>default_rescue</code>	16
Kivételek	16
Magasabb rendű függvények	16
Call agent (parciális függvény alkalmazás)	16
Lambda, inline-agent	17
Osztály vázlat	17
Minta ZH megoldása	18
Feladateleírás	18
Irányított gráfot reprezentáló adattípus	18
Irányítatlan gráf	19
Unió	19
Bejárás	19
Megoldás	19
DIGRAPH osztály	19
GRAPH osztály	21
Főprogram, minimális tesztelés	21

Gyakorlati jegyzet

- [Standard könyvtár dokumentáció](#)

Általános információk, elvek

- Minden kód osztályokban található
 - Főprogram is egy osztály, fordításkor kell megadni
 - Nincs statikus rutin, attribútum
- Szerződéses futási időben kerülnek ellenőrzésre
 - Valójában a szerződések csak szintaxikus cukorkák assertion-ök beszúrására
- Szerződésekkel nem tudunk mindent kifejezni
 - pl. `gcd(a,b)=gcd(Result,number)` szerződés végtelen rekurzióhoz vezetne
 - Ha ki is tudunk valamit fejezni, lehet, hogy nem jó ötlet, mert túl lassú lesz tőle a program

Vizsgán ezekre oda kell figyelni

- Adattaghoz tilos “triviális” gettert írni
 - Indoklás: adattagot csak az ő osztálypéldánya tudja módosítani, senki más. Tehát korlátozni a láthatóságát és gettert készíteni hozzá hülyeség.
 - Viszont néha szükséges, hogy a belső állapot ne szökhessen ki.
- Ciklushoz kötelező invariáns és termináló függvényt írni
- `attached` kulcsszót mindenhol explicit ki kell írni
- Frame condition-t is írjunk `ensure`-be

Saját tippjeim:

- Ha valami nem működik és a hibaüzenet teljesen értelmetlennek tűnik, akkor keressük a hibát az alosztályokban (pl. ideiglenesen töröljük ki őket - megszűnik így a hiba?)
 - Gyakori hiba: hiányzik alosztályból egy `creation procedure`

Szintaxis, konvenciók

- Pontosvessző opcionális
- Kis- és nagybetűk:
 - Nem számít a különbség kulcsszavakban, azonosítóiban, viszont vannak konvenciók
 - Kulcsszó: kisbetű

- Egyéb foglalt szó: első betű nagy
 - * pl.: `Result`, `Current`, `Void`, `True`, `False`
- Konstans: `Line_width: INTEGER = 256`
- Osztálynév: csupa nagybetű
- Többi azonosító: kisbetű
- Tagolás: alsó vonás (`_`)
- Üres paraméterlistát (`()`) nem írjuk ki
 - Így attribútumok és paraméter nélküli függvények használata megegyezik

Más nyelvek	Eiffel
<code>this</code>	<code>Current</code>
<code>return ...</code>	<code>Result := ...</code>
<code>null</code>	<code>Void</code>

Láthatóság

- Publikus: `{ANY}` vagy semmi
- Protected-szerűség: `class C feature {C} f ... end`
- Titkos: `{NONE}`
 - Nem ugyan az, mint a privát más nyelvekből:
 - * Ez nem osztályra, hanem objektumra privát (azaz csak saját magadra lehet meghívni)
 - * Alosztályban el lehet érni a szülőosztály titkos dolgait

Alapvető műveletek, operátorok, konstansok

- `True`, `False`
- `not`, `and`, `or`, `xor`, `implies`
- Lusta operátorok: `and then`, `or else`
- Egész osztás: `//`; osztási maradék: `\`; hatványozás: `^`
- `<`, `<=`, `>`, `>=`
- Referencia összehasonlítás: `=`, `/=`
- Belső tartalom összehasonlítás: `~`, `/~`
- Tömb indexelés: `[,]`
- Típusozott literál: `{REAL_32} 3.14`
- Egész intervallumon iterálás: `across n |...| m as i ...`
- Standard kimenetre írás
 - Objektum kiírása: `print(something.out)`
 - Újsor: `print("%N")`

Void-biztonság, attached, detachable

- Csak referencia típusoknál kell erre figyelni
- Void-mentes változó: `attached TÍPUS`
 - Ez az alapértelmezett, de explicit ki kell írni a vizsgán
- Void-ot tartalmazni képes változó: `detachable TÍPUS`

```

local
d: detachable MYCLASS
do
d.something() --Hiba: d egy detached típusú változó

if attached d as ad then
  ad.something()
end

--Check: szerződésszegés történik, ha nem teljesül a feltétel

```

```

--Emiatt egyes esetekben jobb, mint az if: itt nincs else ág
check attached d as ad then
  ad.something()
end

if d /= Void then
  d.something() --CSAK paraméter/lokális változó esetén tudja a fordító, hogy d itt már attached
end
end

```

Osztályok

```

(expanded) class <OSZTÁLYNÉV>
feature <RUTINOK, ATTRIBÚTUMOK>
end

```

- Referencia típus (class)
 - Indirekt hozzáférés, aliasing
 - Heap memória, szemétygyűjtés
 - Polimorf változók, dinamikus kötés
 - Üres referencia: Void
 - Megosztással került átadásra (call-by-sharing)
 - * Vigyázni kell, mert például több paraméter is mutathat ugyan arra az objektumra, példáulra Current-re
 - Belső állapotot könnyű kiszivároztatni
 - * STRING esetén érdemes a twin feature-t használni ennek elkerülése érdekében
- Kifejtett típus (expanded class)
 - Direkt módon érhető el, kifejtésre kerül a tároló osztályba/verembe
 - Érték szerint kerül átadásra (call-by-value)
 - Kifejtett típusnak nincs alosztálya

Rutinok: eljárások és függvények

```

--Eljárás: nincs visszatérési érték, csak mellékhatást okoz
-- Neve legyen cselekvést leíró ige, pl. divide_by
my_procedure(param_a: STRING; param_b: INTEGER)
local
  temp: INTEGER --Nullaszerű értékre inicializálódik
do
  --Paraméterek értéke nem írható felül
  print("Hello World!%N")
end

--Függvény: van visszatérési érték
-- Neve legyen eredményre utaló névszó, pl. divided_by
my_function(param_b: INTEGER): INTEGER
do
  Result := param_b + 1
end

--Once rutin: törzs csak egyszer fut le, az eredmény elmentésre kerül és későbbiekben
-- csak az kerül visszaadásra. Probléma: ha más paraméterekkel kerül meghívásra
-- másodjára, akkor is az eredetileg kiszámolt értéket adja vissza.
feature
  talk(kind_of_ignored: INTEGER): INTEGER
once
  Result := kind_of_ignored

```

```
end
```

Operátorok

```
feature --1 formális paraméter => bináris infix operátor: (lehet bármilyen karaktersorozat)
  multiplied_by alias "*" (other: attached FRACTION): attached FRACTION
  do create Result.set(num * other.num, ...) end
```

--Használat:

```
x, y, z: FRACTION
```

```
z := x * y --Jelentése: x.multiplied_by(y)
```

```
feature --0 formális paraméter => Unáris prefix operátor: -
  negated alias "-": attached FRACTION
  do create Result.set (-num, den) end
```

--Használat:

```
x, y: FRACTION
```

```
y := -x --Jelentése: x.negated
```

[] alias-szal rendelkező operátor akárhány aritású lehet, használat: x[a,b,c]

Attribútumok (adattagok)

Attribútumnak értéket adni csak az osztálynak a saját **példánya** tud. Következmény:

- Láthatóságot általában nem szükséges korlátozni. Gettert nem kell írni.
- Kivülről módosításhoz egy setter rutin szükséges.

```
feature -- egyszerű attribútum
  --Nullaszerű értékre inicializálódnak
  name: STRING --Kezdeti érték: Void
  x, y: INTEGER --Kezdeti érték: 0
```

```
feature -- konstans
  Answer: INTEGER = 42
```

```
feature -- inline inicializáció: nem használjuk, mert az osztály invariáns ellenőrzés után fut csak le
  test: INTEGER attribute Result := 123 end
```

```
feature -- szintaktikus cukorka: := operátor támogatása
  name: STRING assign set_name
  set_name(new_name: STRING)
  do name := new_name.twin end
  set_friends_name(friend: PERSON)
  --Ez történik valójában: friend.set_name(name)
  do friend.name := name end
```

Példányok létrehozása, creation procedure

```
class PERSON
  create set_name --Opcionálisan megadható láthatóság, pl. protected: create {PERSON} set_name
  --{NONE} láthatóságú feature lehet (publikus) creation procedure: független a láthatóságuk
  feature set_name(name: attached STRING) do ... end
  --Gyakori creation procedure: saját osztálypéldányt paraméterként
```

--Használat:

```
local
```

```
  bob: attached PERSON
```

```

  alice: ANY
do
  create bob.set_name("Bob") -- utasítás
  bob := create{PERSON}.set_name("Bob") -- kifejezés
  create{PERSON} alice.set_name("Jack") -- változónak alosztály értéket adunk

```

Szülőosztály creation procedure-jei nem lesznek az alosztálynak is creation procedure-jei: minden osztálynak saját magának deklarálni kell, hogy mely rutinok a creation procedure-ök (kivétel: `default_create`).

default_create

Osztály nem ad meg creation procedure-t \implies az ANY osztályból megörökölt azon művelet egy engedélyezett creation procedure lesz, amit az ANY osztályban `default_create`-nek neveznek. Ez a creation procedure lehet, hogy át lett nevezve.

```

--default_create-et (eredeti vagy új nevén) nem kell kiírni meghíváskor
class TEST
inherit
  ANY rename default_create as make end
end

create {TEST} --create {TEST}.make

--default_create "letiltásra" kerül, ha van másik creation procedure, de explicit újra lehet engedélyezni
class TEST
create make, default_create
feature make do end
end

--default_create felüldefiniálása
class MYCLASS
inherit ANY redefine default_create end
create default_create
feature default_create do ... end
end

```

Szerződések

Három fő szerződés van:

- Osztály invariáns (`invariant`)
 - Létrehozó művelet beállítja az invariáns
 - Műveletek megőrzik az invariánst
- Rutin előfeltétel (`require`)
 - Osztály mezőit és a rutin paramétereit használhatja
- Rutin utófeltétel (`ensure`)

```

class TEST
feature
  counter: INTEGER
  nullable: detachable STRING
feature
  div(a: INTEGER, b: INTEGER): INTEGER
    require
      division_by_zero: b /= 0
    do
      counter := counter + 1
      Result := a // b
    ensure
      counter_incremented: counter = old counter + 1

```

```

    Result = a // b --Címke megadása nem kötelező
    frame: strip(counter) ~ old strip(counter)
end
invariant
  some_check: counter >= 0
  --detachable attribútum esetén körülményes a tud lenni check megfogalmazás:
  another: nullable != Void implies attached nullable as a_n and then a_n.count > 0
end

```

old operátor

- Csak utófeltételben szerepelhet
- old xyz azt az értéket adja vissza, ami xyz értéke volt, a rutin futásának legelején

Frame problem, frame rule, frame condition

- Utófeltételbe ha nem írjuk bele, hogy nem változik meg minden, akkor nem is lenne szabad feltételezni
- Egy megoldás: strip(a,b,c) segítségével ellenőrizhető, hogy a,b,c mezők kivételével minden ugyan az maradt-e Current-ben
- Példa: frame: strip(balance) ~ old strip(balance)

Ellenőrzések do klózban

Futási idejű assertion-ök hozhatók létre az alábbi módon. A program végrehajtás csak akkor halad tovább, ha az állítások igazak.

```

check size <= capacity end

check
  size_is_not_too_large: size <= capacity
end

```

Felüldefiniált rutin elő- és utófeltétele

```

class IDOPONT
inherit
  DATUM redefine from_array end
create
  from_array
feature
  from_array(arr: attached ARRAY[INTEGER])
  require else --Előfeltétel gyengítése: vagy az eredetinek, vagy ennek kell teljesülnie
    arr.count = 5
  do ...
  ensure then --Utófeltétel szigorítása: mind a réginek, mind ennek teljesülnie kell
    ora = 0 or else ora = arr[arr.lower+3] \ \ 24
    perc = 0 or else perc = arr[arr.lower+4] \ \ 60
  end
end
end

```

- Nincs require \equiv require True
- Nincs require else \equiv require else False
- Nincs ensure \equiv ensure True
- Nincs ensure then \equiv ensure then True

Hoare-hármas

- {előfeltétel} program {utófeltétel} szintaxisban megadott elő- és utófeltétel
- Továbbiakban ezt a jelölést fogjuk alkalmazni

Utasítások

Elágazás

```
if ev \\ 400 = 0 then Result := True
elseif ev \\ 100 = 0 then Result := False
elseif ev \\ 4 = 0 then Result := True
else Result := False
end
```

Sokágú elágazás: inspect

```
inspect honap
  when 1,3,5,7,8,10,12 then Result := 31
  when 4,6,9,11 then Result := 30
  when 2 then
    if szokoev then Result := 29
    else Result := 28
    end
end
```

Általános ciklus

```
gcd(a, b: INTEGER): INTEGER
local
  number: INTEGER
do
  from
    Result := a
    number := b
  invariant
    0 < Result; 0 < number
    -- gcd(a,b) = gcd(Result,number) --This is logically correct, but would result in infinite recursion
  until
    Result = number
  loop
    if Result > number
    then Result := Result - number
    else number := number - Result
    end
  variant
    Result + number
  end
end
```

Ciklus szerződése

```
from INIT -- Inicializáció; utasítás(ok)
invariant INV -- Végig igaz állítás; A -> BOOLEAN
until COND -- Terminálási feltétel; A -> BOOLEAN
loop BODY -- Ciklustörzs; utasítás(ok)
variant VAR -- Iterációnként csökkenő érték; A -> INTEGER
end
--"A" jelentése: állapottér, azaz paraméterek, lokális változók, adattagok értékei
```

Egy ciklus megfelel a szerződésnek, ha:

- $\{True\} INIT \{INV\}$
- $\{INV \wedge \neg COND\} BODY \{INV\}$

- $INV \implies VAR \geq 0$
- $\forall v: \{INV \wedge \neg COND \wedge VAR = v\} BODY \{VAR < v\}$

Bejáró ciklus

```
across
  <<1969, 7, 20, 20, 17, 40>> as i
loop
  print(i.item.out) --i egy iterátor féleség, i.item tartalmazza az értéket
  print("%N")
end
```

Kvantálás

Hasznos például elő- és utófeltételek írásánál, hiszen ezek logikai kifejezések.

```
--Univerzális kvantálás
mybool := across <<7, 20, 20, 17, 40>> as i all i.item > 0 end

--Egzisztenciális kvantálás
mybool := across <<7, 20, 20, 17, 40>> as i some i.item = 17 end
```

Öröklődés

- Osztályból való öröklődés letiltható a `frozen` kulcsszóval: `frozen class XYZ ...`
 - Valójában csak altípusosságot tiltja meg: privát öröklődést nem
- Alosztályban el lehet érni a szülőosztály titkos dolgait

```
class SAVINGS_ACCOUNT
inherit ACCOUNT
  rename make as make_account --Rutin átnevezhető
end
create
  make_account --Feature a szülőosztályból származik
  make
feature
  --Újabb make_account deklaráció névütközést okozna
  interest: INTEGER assign set_interest
  make(id_, interest_: INTEGER)
  require
    non_negative_interest: interest_ >= 0
  do
    make_account(id) --Szülő "konstruktor" hívás
    set_interest(interest_)
  require
    id = id_; interest = interest_
  end
...
invariant
  interest >= 0
end
local
  a: attached ACCOUNT
do
  create {SAVINGS_ACCOUNT} a.make(42, 3)

  a.set_interest(10) -- fordítási hiba
```

```

if attached {SAVINGS_ACCOUNT} a as sa then
  sa.set_interest(10)
end

check attached {SAVINGS_ACCOUNT} a as sa then
  sa.set_interest(10)
end

```

Rutinok felüldefiniálása

- Jelezni kell a felüldefiniálást: `inherit XYZ redefine abc end`
- Felüldefiniálás letiltható a `frozen` kulcsszóval: `feature frozen xyz ...`
- Megörökölt (“előző”) implementáció meghívható a `Precursor` kulcsszóval
 - Több előző implementáció közül a `Precursor {CLASSNAME}` szintaxis segítségével kiválasztható, hogy melyik `Precursor`-re gondolunk

Kifejtett osztályok öröklődése

- Kifejtett osztály lehet altípusa egy referencia típusnak
- Kifejtett osztályból való öröklődés viszont **nem** vezet be altípusosságot
 - Akkor se, ha egy referencia típusú osztály örököl a kifejtett osztályból

Privát öröklődés

- Csak a kód öröklődik
 - **Nem** vezet be altípusosságot
 - Gyakorlatilag bemásolásra kerül a kód:
 - * Publikus feature publikus marad
 - * Once rutin így “többször” le fog futni (egyszer az eredeti osztályban, egyszer ebben az új osztályban)
- Lehet `frozen` osztályból privát módon örökölni

Örökölt rutinok láthatóságának megváltoztatása

```

class QUEUE[T]
inherit {NONE} SEQUENCE[T]
  export {ANY} hiext, lov, lorem, size;
    {QUEUE} all -- e.g. hirem, hiv, loext
end
end

```

Absztrakt osztályok, rutinok

- `deferred` osztályban deklarálható implementáció nélküli (`deferred`) rutin
- `deferred` osztályból öröklődés során...
 - Vagy meg kell adni a hiányzó implementációt
 - Vagy az öröklő osztálynak is `deferred`-nek kell lennie
- Hiányzó (`deferred`) implementáció megvalósítása esetén nem kell `redefine`-t írni
- Szülőosztály rutin implementációja eldobható a `redefine` kulcsszóval, ilyenkor `deferred` rutin lesz belőle
 - Többszörös öröklődésnél lesz elsősorban hasznos, hogy ne legyen egy rutinhoz több implementáció
- A `deferred` a `do`-t helyettesíti; elő- és utófeltétel megadható attól még

```

deferred class ANIMAL
feature
  talk: STRING deferred end
end

--Egy lehetséges megvalósítás
class CAT

```

```

inherit ANIMAL
feature
  talk: STRING do Result := "Miaow" end
end

--Alternatív megvalósítás
class CAT
inherit ANIMAL redefine default_create end
feature
  talk: STRING
  default_create do talk := "Miaow" end
end

```

Többszörös öröklődés

- Lehet publikus és privát módon is több osztályból örökölni
 - Egy osztályt akár többször is meg lehet örökölni
- Amennyiben több feature...
 - Különböző néven öröklődés: több feature lesz
 - Azonos néven öröklődik, több irányból, azonos implementációval: egy feature lesz (automatikus join)
 - * Csak akkor, ha ezek a feature-ök azonosak, azaz diamond inheritance esetén
 - Azonos néven öröklődik, több irányból, eltérő implementációval: nem lehet join-olni
 - Azonos néven öröklődik, több irányból, egy kivétellel mindegyik **deferred**: egy feature lesz (automatikus join)

Névütközések feloldása

- `rename xyz as abc`: két külön feature-t készítünk az ütköző feature-ökből
- `undefine xyz`: az egyik implementációt eldobjuk, így egy feature lesz
- `redefine xyz`: mindegyik implementációt eldobjuk és új implementációt adunk meg, így egy feature lesz

```

class STATEMENT -- inherits is_equal from ANY
  feature is_right: BOOLEAN
end

class PARTY -- inherits is_equal from ANY
  feature is_right: BOOLEAN
end

--Névütközés, fordítási hiba
class CAMPAIGN_PROMISE
inherit STATEMENT PARTY
end

--Öröklődés során átnevezés
class CAMPAIGN_PROMISE
inherit
  STATEMENT rename is_right as holds end
  PARTY
end --is_equal: két irányból azonos implementáció => join

```

Dinamikus kötés, late binding, dynamic dispatch

```

class ALPHA
feature test: INTEGER do Result := 1 end
end

class BRAVO

```

```
feature test: INTEGER do Result := 2 end
end
```

```
class TEST
inherit
  ALPHA rename test as t end
  BRAVO
end
```

```
--Mi fut le?
local
  a: attached ALPHA
  b: attached BRAVO
  y: attached TEST
do
  create y; a := y; b := y
  print(a.test) --ALPHA#test = TEST#t
  print(b.test) --BRAVO#test = TEST#test
  print(y.test) --BRAVO#test = TEST#test
  print(y.t)    --ALPHA#test = TEST#t
```

Diamond inheritance esetén a `select` klóz segítségével meg kell mondani, hogy melyik “oldalon” található implementáció fusson le.

```
deferred class BASE
feature test: INTEGER deferred end
end
```

```
class ALPHA
inherit BASE
feature test: INTEGER do Result := 1 end
end
```

```
class BRAVO
inherit BASE
feature test: INTEGER do Result := 2 end
end
```

```
--Megtartjuk a feature mindkét implementációját, de az egyiket átnevezzük
```

```
class TEST
inherit
  ALPHA rename test as t end
  BRAVO select test end --Kötelező a select klóz
end
```

```
--Mi fut le?
local
  x: attached BASE
  a: attached ALPHA
  b: attached BRAVO
  y: attached TEST
do
  create y; x := y; a := y; b := y
  print(x.test) --BRAVO#test = TEST#test
  print(a.test) --BRAVO#test = TEST#test (!!!)
  print(b.test) --BRAVO#test = TEST#test
  print(y.test) --BRAVO#test = TEST#test
```

```
print(y.t)      --ALPHA#test = TEST#t
```

Ismételt öröklődés

```
class BINTREE[T]
inherit CELL[T] -- provides item and put
inherit {NONE}
  LINKED
    rename
      next as left,
      set as set_left
    end
  LINKED
    rename
      next as right,
      set as set_right
    end
create put
end
```

Variancia

- detached helyett attached használata: a lehetséges értékek szűkítésének felel meg (kovariancia)
- Kapcsolt típus: like Current vagy like featureName
 - Ennek használatával könnyen tudunk kovariáns rutinokat készíteni, amelyek altípusra is helyesen működnek
 - A like kulcsszó az attached/detachable tulajdonságot is “lemásolja”, de ez felülírható (pl. detachable like featureName)
- Függvénytípusok zárójelezése: $A \rightarrow (B \rightarrow C) \Leftrightarrow A \rightarrow B \rightarrow C$
- Függvénytípusok altípusossága:
 - $A <: A', B' <: B$
 $\Rightarrow A' \rightarrow B' <: A \rightarrow B$
 - $A <: A' \wedge B <: B' \wedge C' <: C$
 $\Rightarrow A' \rightarrow B' \rightarrow C' <: A \rightarrow B \rightarrow C$
 - $A' <: A \wedge B <: B' \wedge C' <: C$
 $\Rightarrow (A' \rightarrow B') \rightarrow C' <: (A \rightarrow B) \rightarrow C$

Liskov helyettesítési elvet nem sértő variancia

- Kovariáns eredmény típus
- Kovariáns utófeltétel
- Kontravariáns paraméter típus
 - Eiffel: csak kovariáns módon változtatható
- Kontravariáns láthatóság
 - Eiffel: kovariáns módon is változtatható
- Kontravariáns előfeltétel

Ahol nincs ellenkezőleg jelezve, arra van lehetőség Eiffelben.

Kovariáns paraméter példa

```
deferred class ANIMAL
feature feed( f: attached FOOD ) deferred end
end

class CAT
inherit ANIMAL
feature feed( m: attached MILK ) do ... end
end
```

```

--Előny:
(create {CAT}).feed( (create {GRASS}) ) -- fordítási hiba

--Probléma: ("polymorphic CAT-call")
local
  a_cat: CAT
  some_grass: GRASS
  polymorphic: ANIMAL
do
  create a_cat; create some_grass
  a_cat.feed( some_grass ) -- fordítási hiba
  polymorphic := cat
  polymorphic.feed( some_grass ) -- sikeres fordítás :(

```

CAT: Changed Availability or Type

Problémát okozhat, ha leszármaztatásnál...

- Vagy örökölt feature-nek csökken a láthatósága
- Vagy mezőnek/rutin paraméternek szűkül a típusa

Generikus osztály

```

class ARRAY [G] --Típusparaméter
...

```

- A típusparaméter általában kovariáns
 - Statikusan nem biztonságos a típusrendszer
 - `class STACK[T] ==> STACK[INTEGER] <: STACK[ANY]`
- Kivétel: frozen típusparaméter esetén invariáns
 - `class STACK[frozen T]`

```

VECTOR[G -> ADDABLE] -- G-nek ADDABLE alosztályának kell lennie (upper bound)
HASHTABLE[K -> HASHABLE, V] -- több típusparaméter
VECTOR[G -> {ADDABLE, HASHABLE}] -- több megszorítás

```

```

VECTOR[G -> ADDABLE create make end] --G származzon le az ADDABLE osztályból és
-- tegye az ADDABLE osztályban található make feature-t egy creation procedure-ré.
-- Lehet, hogy G-nek megfelltetett osztályban az ADDABLE-ben make-nek hívott feature-t
-- már nem make-nek hívják, de ez nem probléma, mi make-ként fogunk hivatkozni rá.

```

```

VECTOR[G -> ADDABLE rename add as plus end] -- átnevezés
VECTOR[G -> {A rename v as w, B}] -- átnevezés, hogy A és B között ne legyen névütközés
--A rename valójában nem megszorítás: nem befolyásolja a típusparaméter helyére
-- helyettesített osztályt. Az az osztály lehet, hogy névütközés esetén például
-- `A rename v as w` helyett a más nevet ad `v`-nek, vagy inkább a `B` osztály `v`
-- metódusát nevezi át a névütközés feloldása érdekében.

```

```

VECTOR[frozen G -> ADDABLE] -- invariant generic param

```

Tuple

```

t2: TUPLE[INTEGER,INTEGER]
t2 := [1,3]
t2.item(0) --ANY típusú, hiszen az item rutin így van definiálva

t3: TUPLE[i,j: INTEGER; r:REAL] --Rekord féleség, név rendelhető komponenshez

```

```
t3 := [1,3,0.0]
t3.i --INTEGER típusú
```

```
TUPLE[INTEGER,STRING] <: TUPLE[INTEGER,ANY] <: TUPLE[INTEGER] <: TUPLE[ANY] <: TUPLE
```

Egyenlőségvizsgálat és másolás

`is_equal`-t (`~`-t) és `copy`-t lehetőségünk van felüldefiniálni. Alapértelmezetten a sekély implementációval ekvivalensek.

A sekély egyenlőségvizsgálat az objektum mezőket `=`-vel hasonlítja össze. A mély egyenlőségvizsgálat a mezőket mély egyenlőségvizsgálattal hasonlítja össze.

A `clone` egy elavult művelet, helyette a `twin` használandó.

Egyedi (<code>~</code>)	Sekély	Mély
<code>is_equal(b)</code>	<code>frozen standard_is_equal(b)</code>	<code>frozen is_deep_equal(b)</code>
<code>frozen equal(a,b)</code>	<code>frozen standard_equal(a,b)</code>	<code>frozen deep_equal(a,b)</code>
<code>copy(b)</code>	<code>frozen standard_copy(b)</code>	<code>frozen deep_copy(b)</code>
<code>frozen twin</code>	<code>frozen standard_twin</code>	<code>frozen deep_twin</code>
<code>frozen clone(b)</code>	<code>frozen standard_clone(b)</code>	<code>frozen deep_clone(b)</code>

```
a.copy(b)      -- a-ba másoljuk b-t
a := b.twin     -- másolat b-ről
a := clone(b)  -- másolat b-ről (obsolete)
```

Referencia típus egyenlőségvizsgálata

- Azonosság (referencia összehasonlítás): `a = b`
- Tartalmi egyenlőség, programozó által definiálva: `a ~ b`
- `=` \subseteq `standard_equal` \subseteq `equal` \equiv `~`
- `=` \subseteq `standard_equal` \subseteq `deep_equal`

Kifejtett típus egyenlőségvizsgálata

- Tartalmi egyenlőség, programozó által definiálva: `a ~ b` vagy `a = b` (kettő megegyezik)
- `standard_equal` \subseteq `=` \equiv `equal` \equiv `~`
- `standard_equal` \subseteq `deep_equal`

Konverzió

```
class FRACTION
create
  from_integer
convert
  from_integer({INTEGER}), --konverzis eljárás: más típus -> saját típus
  from_array({ARRAY[INTEGER]}),
  to_real:{REAL_64} --konverzis függvény: saját típus -> más típus
feature
  from_array(arr: attached ARRAY[INTEGER])
    require arr.count = 2
  do ... end
  ...
end

--Használat:
f: attached FRACTION
r: REAL_64
f := 3 -- create f.from_integer(3)
```

```
f := <<3,1>> -- create f.from_array(<<3,1>>)
r := f -- f.to_real
```

Kivételkezelés

```
feature
my_routine
  require
    ... --Előfeltétel egy retry során nincsen ellenőrizve
  local --Lokális változó megőrzi értékét egy retry során
    already_tried: BOOLEAN
  do
    if not already_tried then
      --Normal operation
    else
      --Alternative operation
    end
  rescue --Ez fut le, ha kivétel lépett fel (ez van try-catch helyett)
    if not already_tried then
      already_tried := True
      retry --Csak rescue-ban lehet
      --Akár végtelenszer is lehetne retry-olni
    end --Ha nem retry-olunk: feljebb terjed a kivétel a stack-en
  end
```

Kivételkezelés fajtái

- Organized panic
 - rescue egy retry nélkül fejeződött be
 - Az utófeltételt nem teljesül
 - Az osztályinvariánst helyreállítottuk
 - A kivétel feljebb terjed a stack-en
- Újrapróbálkozás
 - Törzs újra le fog futni
 - Az előfeltételt a rescue-ban helyreállítottuk

default_rescue

- ANY-ból mindenki megörökli
- Felüldefiniálható, de nem szerepelhet benne retry
 - Megjegyzés: át is nevezhető a default_rescue
- Alapból nem csinál semmit
- Ez fut le, ha nincs egy rutinhoz explicit megadva egy rescue klóz

Kivételek

```
class CONNECTION
inherit EXCEPTIONS --Így elérhető a `raise` rutin
...
... raise( "Communication_failure" ) ... --Kivételek szöveges üzenetek
...
end
```

Magasabb rendű függvények

Call agent (parciális függvény alkalmazás)

Hivatkozás egy függvényre, illetve annak parciális alkalmazása. A hívás nem hajtódik végre, csak hivatkozunk rá.


```

class TEST
feature
  f(s: STRING, i: INTEGER): REAL do ... end
  g(s: STRING) do ... end

  test(obj: TEST)
  do
    ... = agent f --Típus: FUNCTION[STRING, INTEGER, REAL]
    ... = agent Current.g --Típus: PROCEDURE[STRING]
    ... = agent obj.f("abc", ?) --Típus: FUNCTION[INTEGER, REAL]
    ... = agent {TEST}.f(?, 42) --Típus: FUNCTION[TEST, STRING, REAL]

    --FUNCTION és PROCEDURE ösosztálya: ROUTINE
    --A függvényhez tartozó osztálypéldány ha nincs megadva, akkor úgy működik, mint egy paraméter
    --FUNCTION: utolsó típusparaméter a visszatérési érték típusa
  end
end

foreach(f: PROCEDURE[REAL]; a: ARRAY[REAL])
do
  across a as i loop p(i.item) end --agent meghívása: zárójelekkel
end

```

Lambda, inline-agent

- Rutin szintaxissal megegyezik, csak név helyett **agent** kulcsszót tartalmaz
 - Lehet elő- és utófeltétele, lokális változója
 - Használhatja a befoglaló osztály feature-jeit
- Nem használhatja a befoglaló rutin lokális változóit

```
agent(x,y: INTEGER): BOOLEAN do Result := x-y > 5 end
```

Osztály vázlat

```

(expanded | deferred | frozen) --Value type? Abstract? Disallow subtyping?
class <CLASS_NAME> [<TYPE_PARAMETER> -> <TYPE_PARAMETER_UPPER_BOUND>]

inherit --Conforming inheritance
  <PARENT_CLASS> rename <old_name> as <new_name> -- e.g. to fix name clashes
  redefine <routine_name> -- define new implementation for a non-deferred routine
  undefine <routine_name> -- drop a routine's implementation (make it deferred)
  select <routine_name> -- select feature for dynamic dispath in case of diamond inheritance
  export {VISIBILITY} <routine_name> -- change feature's visibility
  {ANOTHER_VISIBILITY} <routine_name>
  end
  <PARENT_CLASS>
  ...

inherit {NONE} --Inherit code without creating subtype relation
  ...

create <routine_name>, <another_routine_name>
create {<CLASS_NAME>} <routine_name> --Only accessible by the class and its subclasses
convert <routine_name>

feature --Public attributes
  <Constant_name>: <TYPE> = <LITERAL>
  <variable>: <TYPE>

```

```

<variable>: <TYPE> assign <variable_setter> --Usage: xyz.variable := ... or xyz.variable_setter(...)

feature --Public functions/procedures (routines)
  (frozen) --Disallow redefinition?
  <procedure> | <procedure>(<param1, param2>: <TYPE12>; <param3>: <TYPE3>) | <function>: <RETURN_TYPE>
    require <precondition: logical expression>
    local <temporal variables>
    (do | once) <routine body> --Once: body is only run once, result is cached (params -> dangerous)
    ensure <postcondition: logical expression>
    rescue <exception handling>
    end

  <routine_name>
    require <...>
    deferred --Abstract routine can only be defined in an abstract class
    ensure <...>

  <redefined_routine_name>
    require else <...> --Default implementation: require else False
    do ... --Previous implementation is accessible via Precursor(...)
    ensure then <...> --Default implementation: ensure then True

  <function_name> alias "binary_infix_operator" (<param>: <TYPE>): <RETURN_TYPE>
  ...

  <function_name> alias "unary_prefix_operator" : <RETURN_TYPE>
  ...

  <function_name> alias "[]" (<any number of parameters>): <RETURN_TYPE> (assign <procedure_name>)
  ...

feature {NONE} --Features only accessible by the current object instance (in this class or a subclass)
  ...

feature {<CLASS_NAME>} --Features which are only accessible by the class and its subclasses
  ...

invariant <logical expression> --Must be true after any creation procedure finishes
end

```

Minta ZH megoldása

Az alábbi feladat Kozsik Tamás honlapján publikusan megtalálható: peldazh.html

Feladatléírás

Írányított gráfot reprezentáló adattípus

Készítsünk egy DIGRAPH osztályt, amely egy irányított gráfot ábrázol. A gráf csúcspontjaiban tárolt elemek típusát sablonparaméterként kapjuk. Ez az elemtípus HASHABLE kell legyen. A gráfot ábrázoljuk hasítótáblával a következőképpen: a kulcsok értelemszerűen a csúcsok lesznek, egy kulcshoz pedig azon csúcsok halmazát rendeljük, ahova vezet él. Használjuk az Eiffel beépített HASH_TABLE és ARRAYED_SET típusát.

- Valósítsuk meg a legfontosabb gráfműveleteket: csúcs hozzáadása, él létrehozása, és egy feature-t, ami eldönti, hogy van-e él két csúcs között.
- Biztosítsuk azt, hogy élet csak akkor tudunk létrehozni, ha a végpontjai ebben a gráfban vannak.

- A gráf a creation feature-ét örökölje a hasítótáblából (csak nevezzük át `init`-nek!), amely paraméterként a hasítótábla kapacitását kapja.

Irányítatlan gráf

Készítsünk egy olyan `GRAPH` osztályt, amelyet a `DIGRAPH` osztályból származtatunk, és egy irányítatlan gráfot valósít meg. Ezt úgy érhetjük el, hogy az élet mindkét irányba felvesszük.

Unió

Készítsünk olyan creation feature-t a `DIGRAPH` osztályban, amely két gráfot kap paraméterként, és egy olyat hoz létre, amely a két paraméter uniója. A típushelyesség biztosítása mellett (azaz egy irányítatlan gráf nem kaphat irányított gráfot) oldjuk meg, hogy az implementációt ne kelljen felüldefiniálni a `GRAPH` osztályban.

Bejárás

Készítsünk egy feature-t, amely a gráf mélységi bejárását végezi el. A feature egy ágenst kap paraméterben, és ezt az ágenst hívja meg minden csúcsra a bejárás során.

Megoldás

DIGRAPH osztály

```
class
  DIGRAPH [T -> HASHABLE]

inherit
  HASH_TABLE [attached ARRAYED_SET[T], attached T]
    rename make as init, count as node_count, has as has_node
    export {HASH_TABLE} all --clients should only use the features declared in the DIGRAPH class
      {ANY} node_count, has_node
    end

create init, union

feature
  add_node(node: attached T)
    require
      node_not_present: not has_node(node)
    do
      extend(create{ARRAYED_SET[T]}.make(0), node)
    ensure
      node_present: has_node(node)
      node_count_increased: node_count ~ old node_count + 1
      edge_count_same: edge_count ~ old edge_count
    end

  add_edge(node_from, node_to: attached T)
    require
      node_from_present: has_node(node_from)
      node_to_present: has_node(node_to)
      edge_not_present: not has_edge_between(node_from, node_to)
    do
      definite_item(node_from).extend(node_to)
    ensure
      edge_present: has_edge_between(node_from, node_to)
      edge_count_increased: edge_count > old edge_count --increases by 2 in GRAPH
      node_count_same: node_count ~ old node_count
```

```

end

has_edge_between(node_from, node_to: attached T): BOOLEAN
require
    node_from_present: has_node(node_from)
    node_to_present: has_node(node_to)
do
    Result := definite_item(node_from).has(node_to)
ensure
    frame: Current ~ old Current
end

edge_count: INTEGER
do
    across Current as i
    from Result := 0
    loop Result := Result + i.item.count
    end
ensure
    result_positive: Result >= 0
    frame: Current ~ old Current
end

depth_first_traverse(root: attached T; visitor: attached PROCEDURE[attached T])
require
    root_present: has_node(root)
local
    discovered: attached ARRAYED_SET[attached T]
    visited_count: INTEGER
    to_visit: attached ARRAYED_STACK[attached T]
    v: attached T
do
    from
        create discovered.make(node_count)
        visited_count := 0
        create to_visit.make(node_count)
        discovered.extend(root); to_visit.extend(root)
    invariant
        node_count >= discovered.count
        discovered.count ~ visited_count + to_visit.count
    until
        to_visit.is_empty
    loop
        v := to_visit.item; to_visit.remove
        visitor(v)
        visited_count := visited_count + 1
        across definite_item(v) as i loop
            if not discovered.has(i.item) then
                discovered.extend(i.item)
                to_visit.extend(i.item)
            end
        end
    loop
    variant
        node_count - visited_count
    end
ensure

```

```

        frame: Current ~ old Current
    end

feature {NONE}
    union(x,y: like Current)
    do
        init(0)
        copy(x)
        across y as key_values loop
            if not has_node(key_values.key) then
                add_node(key_values.key)
            end
            across key_values.item as value loop
                if not has_edge_between(key_values.key, value.item) then
                    add_edge(key_values.key, value.item)
                end
            end
        end
    end
    ensure
        node_count_greater_equal: node_count >= x.node_count and node_count >= y.node_count
        node_count_at_most_sum: node_count <= x.node_count + y.node_count
        edge_count_greater_equal: edge_count >= x.edge_count and edge_count >= y.edge_count
        edge_count_at_most_sum: edge_count <= x.edge_count + y.edge_count
    end

end

```

GRAPH osztály

```

class
    GRAPH [T -> HASHABLE]

inherit
    DIGRAPH [T]
        redefine add_edge
        end

create init, union

feature
    add_edge(node_from, node_to: attached T)
    do
        Precursor(node_from, node_to)
        if node_from /~ node_to then
            Precursor(node_to, node_from)
        end
    end
    ensure then
        edge_present_backward: has_edge_between(node_to, node_from)
    end

end

```

Főprogram, minimális tesztelés

```

local
    g,g2,g3: attached GRAPH[INTEGER]
do

```

```

create g.init(10)
g.add_node(11); g.add_node(12); g.add_node(13); g.add_node(14)
g.add_edge(11, 11); g.add_edge(11, 12); g.add_edge(11, 13); g.add_edge(13, 14)
g.depth_first_traverse(11, agent(v: INTEGER) do print(v); print("%N") end)

print("%N")

--g.add_node(11) --Constraint violation
--g.add_edge(11, 999) --Constraint violation
--g.add_edge(11, 12) --Constraint violation
--print(g.has_edge_between(11, 999)) --Constraint violation

g2 := g
create g3.init(10)
g3.add_node(14); g3.add_node(15)
g.add_edge(14, 15)
create g.union(g2,g3)
g.depth_first_traverse(11, agent(v: INTEGER) do print(v); print("%N") end)
end

```