Interpretação e Compilação de Linguagens de Programação

Licenciatura em Engenharia Informática
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa

João Costa Seco
joao.seco@di.fct.unl.pt
Autores: Luís Caires, João Costa Seco

Unidade 9: Linguagens Orientadas por Objectos

O paradigma de programação dominante é o paradigma orientado por objectos. O conceito de objecto agrega estado e funcionalidade num contexto de visibilidade próprio. Os conceitos básicos para termos uma linguagem orientada por objectos são: Dynamic lookup, Abstraction, Subtyping, Inheritance.

Um objecto pode ser modelado por um registo com constantes, variáveis de estado, métodos (funções), e a definição recursiva de um identificador especial "self" ou "this". Os objectos podem ser criados através de padrões bem definidos em "classes".

- Valores Produto e Produtos Etiquetados
- Tipos Produto (Regras de tipificação)
- Conceitos básicos de objectos e classes
- Linguagens Orientadas por objectos
- Objectos na linguagem CORE
- Semântica por tradução na linguagem CORE
- Classes na linguagem CORE
- Tipificação de objectos
- Subtipificação

Valores Produto

- Correspondem aos records da linguagem Pascal,

```
Type PersonName = packed array[1..20] of char;
     PersonInfo = Record
                    name:PersonName;
                    age: Integer;
                    end;
Var person1: PersonInfo;
```

```
person1.age := 25;
```

- structs da linguagem C,

```
typedef struct { char[20] name; int age; } person_info;
...
person_info p;
p.age := 25;
```

- ou produtos etiquetados na linguagem ML.

```
# type person_info = { name:string; age:int };;
type person_info = { name : string; age : int; }
# fun p -> p.name;;
- : person_info -> string = <fun>
```


Valores Produto

- Um valor de tipo produto, também chamado registo, é um valor estruturado que vários "agrega" vários valores, de tipos possivelmente diferentes, numa só entidade.
- O constructor básico é o operador binário de produto cartesiano

```
# (1,2);;
- : int * int = (1, 2)
# ("ola",("mundo",0));;
- : string * (string * int) = ("ola", ("mundo", 0))
```

- Construção de registos

```
v = [ nome = "rita", idade = 1 ]
c = [ real = 0.5+0.5, imag = 2.0 ]
```

- Manipulação de registos (selecção de campo)

```
v.nome = "rita"
c.real = 1.0
v.cor = ?                (erro: campo não existente)
```


Valores Produto

- **Constructores para as expressões sobre registos:**

newrecord:	(string × EXP) list → EXP
selectfield:	EXP × string → EXP

- **Exemplo de uso**

```
decl
  p1 = [nome="Albert"; QI=var(250)]
in decl
  p2 = [nome="Hulk"; QI=var(50)] in
do
  p2.QI := !(p2.QI)+2
return
  !(p2.QI) + !(p1.QI)
end
end
end
```


Tipos Produto

- As operações disponíveis são a construção de registo e a selecção de campo de registo.
- $\mathrm{Tuple}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n)$:
  É o tipo dos registos com campos $\mathrm{id}_1, \ldots, \mathrm{id}_n$, de tipo $\mathcal{T}1, \ldots, \mathcal{T}n$.

$$\frac{Env \vdash E : \mathrm{Tuple}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n)}{Env \vdash E.\mathrm{id}_j : \mathcal{T}_j} \textbf{(Select)}$$

$$\frac{Env \vdash E_1 : \mathcal{T}_1 \quad \ldots \quad Env \vdash E_n : \mathcal{T}_n}{Env \vdash [\, \mathrm{id}_1 = E_1, \ldots, \mathrm{id}_n = E_n \,] : \mathrm{Tuple}(\mathrm{id}_1{:}\mathcal{T}_1, \ldots, \mathrm{id}_n{:}\mathcal{T}_n)} \textbf{(Record)}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

  decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c **end**

- A expressão seguinte está bem tipificada?

decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c.succ(!c.loc) **end**

Tipos Produto (Quiz)

- Qual o tipo da expressão?

  decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c **end**

$$\dfrac{\dfrac{\dfrac{x{:}\mathbf{int} \vdash x : \mathbf{int} \quad x{:}\mathbf{int} \vdash 1 : \mathbf{int}}{x{:}\mathbf{int} \vdash x{+}1 : \mathbf{int}}}{\varnothing \vdash \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end} : \mathbf{Fun(int)\ int}}}{\dfrac{\varnothing \vdash [\ succ = \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end},\ loc = \mathbf{var}(0)\ ] : ?}{\varnothing \vdash \mathbf{decl}\ c = [\ succ = \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end},\ loc = \mathbf{var}(0)\ ]\ \mathbf{in}\ c\ \mathbf{end} : ?}}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

  decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c **end**

$$\dfrac{\dfrac{\varnothing \vdash \mathbf{fun} \ldots : \mathbf{Fun(int)\ int} \qquad \dfrac{\varnothing \vdash 0 : \mathbf{int}}{\varnothing \vdash \mathbf{var}(0) : \mathbf{Ref(int)}}}{\varnothing \vdash [\ succ = \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end},\ loc = \mathbf{var}(0)\ ] : ?}}{\varnothing \vdash \mathbf{decl}\ c = [\ succ = \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end},\ loc = \mathbf{var}(0)\ ]\ \mathbf{in}\ c\ \mathbf{end} : ?}$$

Tipos Produto (Quiz)

- Qual o tipo da expressão?

  decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c **end**

Resposta: **Tuple**($succ$: **Fun(int)int**, loc : **Ref(int)**)

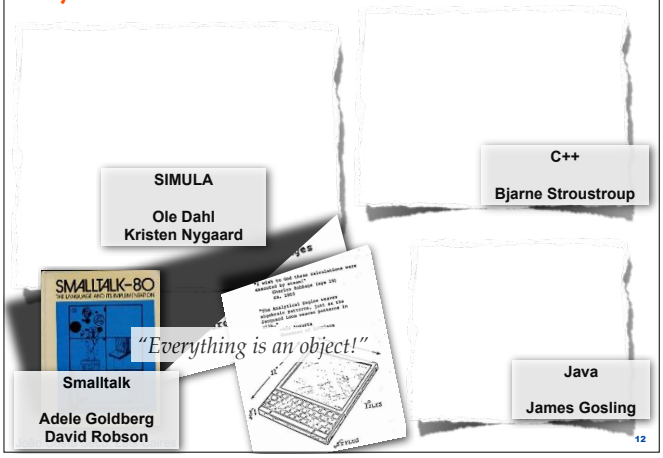
$$\dfrac{\varnothing \vdash [\ \ldots\ ] : \mathbf{Tuple}(succ : \mathbf{Fun(int)int}, loc : \mathbf{Ref(int)}) \qquad \mathrm{c} : \mathrm{Tuple(succ:Fun(int)int,\ loc=Ref(int))} \vdash \mathrm{c} : ?}{\varnothing \vdash \mathbf{decl}\ c = [\ succ = \mathbf{fun}\ x{:}\mathbf{int} \to x{+}1\ \mathbf{end},\ loc = \mathbf{var}(0)\ ]\ \mathbf{in}\ c\ \mathbf{end} : ?}$$

Tipos Produto (Quiz)

- A expressão seguinte está bem tipificada?

decl c = [$succ$ = **fun** x → x+1 **end**, loc = **var**(0)] **in** c.succ(!c.loc) **end**

Objectos e Classes



Simula 67

```
Begin
   Class Glyph;
      Virtual: Procedure print Is Procedure print;;
   Begin
   End;

   Glyph Class Char (c);
      Character c;
   Begin
      Procedure print;
         OutChar(c);
   End;

   Glyph Class Line (elements);
      Ref (Glyph) Array elements;
   Begin
      Procedure print;
      Begin
         Integer i;
         For i:= 1 Step 1 Until UpperBound (elements, 1) Do
            elements (i).print;
         OutImage;
      End;
   End;

   Ref (Glyph) rg;
   Ref (Glyph) Array rgs (1 : 4);

   ! Main program;
   rgs (1):- New Char ('A');
   rgs (2):- New Char ('b');
   rgs (3):- New Char ('b');
   rgs (4):- New Char ('a');
   rg:- New Line (rgs);
   rg.print;
End;
```

$$Z = \sum_{i=1}^{100} \frac{1}{(i+a)^2}$$

```
Real Procedure Sigma (l, m, n, u);
   Name l, u;
   Integer l, m, n; Real u;
Begin
   Real s;
   l:= m;
   While l <= n Do Begin s:= s + u; l:= l + 1;
End;
   Sigma:= s;
End;

Z:= Sigma (i, 1, 100, 1 / (i + a) ** 2);
```

- Baseada no Algol 60
- Objectos, Classes, Subclasses, métodos virtuais, corotinas, simulação de eventos e garbadge collection.
- Call by name

Smaltalk 80

- "Everything is an Object"
- Reflexion mechanism, dynamically typed
- Syntax minimalista
- No real keywords (true, false, nil, self, super, thisContext)
- Construções: message sends, assignment, method return and literais.
- Os programas são editados no próprio ambiente de execução, os sistemas de suporte guardam-se em imagens (cf. máquinas virtuais).

```
3 factorial + 4 factorial between: 10 and: 100
```

```
result := a > b
    ifTrue:[ 'greater' ]
    ifFalse:[ 'less or equal' ]
```

```
| window |
window := Window new.
window label: 'Hello'.
window open
```

```
Object subclass: #MessagePublisher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Smalltalk Examples'
```


Conceitos base de uma linguagem OO

- *Dynamic Lookup*: Quando uma mensagem é enviada a um objecto (executar uma função sobre um conjunto de dados) o código a ser executado depende da implementação concreta e não de uma ligação estabelecida estaticamente com base nos identificadores do programa.
- *Abstracção* (de dados e funcionalidade): Os detalhes de implementação da funcionalidade e da representação de dados são escondidos dentro de um subprograma com um interface bem determinado.
- *Subtyping*: Se um objecto tem a mesma, e possivelmente mais funcionalidades que outro, pode substituí-lo em qualquer contexto.
- *Inheritance*: A capacidade de utilizar um tipo de objectos para definir outros tipos de objectos.

Conceitos base de uma linguagem OO

- *Dynamic Lookup*: Quando uma mensagem é enviada a um objecto (executar uma função sobre um conjunto de dados) o código a ser executado depende da implementação concreta e não de uma ligação estabelecida estaticamente com base nos identificadores do programa.

```
interface I { void doIt(); }

class A implements I {
  void doIt(){ System.out.println("A");}
}

class B implements I {
  void doIt(){ System.out.println("B");}
}

class C implements I {
  void doIt(){ System.out.println("C");}
}
```

VS

```
#define A 0
#define B 1
#define C 2

typedef struct {
  int kind; ...
} I;

void doIt(I o) {
  switch(o.kind) {
  case A: printf("A");
  case B: printf("B");
  case C: printf("C");
  }
}
```


Conceitos base de uma linguagem OO

- *Abstracção* (de dados e funcionalidade): Os detalhes de implementação da funcionalidade e da representação de dados são escondidos dentro de um subprograma com um interface bem determinado.

```
interface Map<K,T> {
  void put(K key, T element);
  T get(K key);
}

class HashMap implements Dictionary {...}

class TreeMap implements Dictionary {...}

class PairList implements Dictionary {...}
```


Conceitos base de uma linguagem OO

- *Subtyping*: Se um objecto tem a mesma, e possivelmente mais funcionalidades que outro, pode substituí-lo em qualquer contexto.

```
interface I { void m1(); }

interface J extends I { void m2(); }

J j = ...
I i = j;
```

Conceitos base de uma linguagem OO

- *Implementation Inheritance*: A capacidade de utilizar um tipo de objectos para definir outros tipos de objectos.

```
class Point {
  int x, y;

  Point(int x, int y) {...}
  ...
}

class ColouredPoint extends Point {
  ...
}
```


Simula

- É uma extensão da linguagem Algol60 em que as classes são procedimentos cujos registos de activação não são destruídos e mantêm válidas as declarações internas (closures).

```
class Point {
  int x, y;

  Point(int x, int y) {...}
  ...
}

class ColouredPoint extends Point {
}
```

access		
real x		
real y		
equals		
distance		

A classe *Counter* em Java (primeira versão)

- Uma classe de objectos "contadores"

```
class Counter implements ICounter {
  int val;
  Counter() { val = 0; }
  void inc() { val = val + 1; }
  int get() { return val; }
}

  ...
  ICounter c = new Counter();
  c.inc();
  c.inc();
  System.out.println(c.get());
```


Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (primeira versão)

```
decl c = [
    val = var(0),
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
in
  c.inc();
  c.inc();
  print(c.get());
end
```


Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (primeira versão)

```
decl c = [
    val = var(0),
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
in
  c.inc();
  c.inc();
  print(c.get());
end
```

o campo val é visível do exterior, é "público"!

val não é visível a partir dos métodos!!!

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

campo val "privado"!

O âmbito de val inclui os métodos do objecto.

Objectos na linguagem CORE (com registos)

- Um objecto da classe Counter, representado na linguagem CORE por um registo contendo variáveis para os membros de dados (segunda versão)

```
decl c =
  decl val = var(0) in
  [
    inc = proc => val := !val + 1 end,
    get = fun => !val end
  ]
  end
in
  c.inc();
  c.inc();
  print(c.get());
end
```

E0
val - l0

E1
c - { inc = (, val := !val+1, E0),
get = (, !val, E0) }

Classes na linguagem CORE (com registos)

- A classe Counter é representada na linguagem CORE por uma função geradora de objectos (primeira versão)

```
decl
  Counter = fun =>
    decl val = var(0) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
    ]
    end
  end
in
  decl c = Counter() in
    c.inc();
    c.inc();
    print(c.get());
  end
end
```


A classe *Counter* em Java (segunda versão)

- Uma classe de objectos "contadores" com constructores...

```
class Counter implements ICounter {
  int val;
  Counter(int n) { val = n; }
  void inc() { val = val + 1; }
  int get() { return val; }
}

  ...
  Counter c = new Counter(0);
  Counter d = new Counter(2);
  c.inc();
  d.inc();
  System.out.println(c.get()+d.get());
```


Classes na linguagem CORE (com registos)

- A classe Counter, é representada na linguagem CORE por uma função geradora de objectos (segunda versão)

```
decl
  Counter = fun n =>
    decl val = var(n) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
    ]
    end
  end
in
  decl c = Counter(0)
       d = Counter(2) in
    c.inc();
    d.inc();
    print(c.get()+d.get());
  end
end
```


Classes na linguagem CORE (com registos)

- Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Nesta codificação não, porquê? dup chama inc...

```
decl
  Counter = fun n =>
    decl val = var(n) in
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => inc(); inc() end
    ]
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```

inc não é visível!!!

Classes na linguagem CORE (com registos)

- Em geral, os métodos de um objecto devem poder chamar-se uns aos outros recursivamente. Nesta codificação não, porquê? dup chama inc...

```
decl
  Counter = fun n =>
    decl val = var(n) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```


Classes na linguagem CORE (com registos)

- A solução consiste na definição de um registo recursivo. Note-se que as referências ao nome "self" apenas ocorrem no corpo de abstracções.

```
decl
  Counter = fun n =>
    decl val = var(n) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
    end
  end
in
  decl c = Counter(0) in
    c.dup();
    ...
  end
end
```

```
E0
val - l0
```

```
E1
self - { inc = (, val := !val+1, E1 ),
         get = (, !val, E1 ),
         dup = (, self.inc(); self.inc(), E1) }
```


Uma sintaxe para Classes

- Classes

```
class
  id1 := E1
  ...
  idp := Ep
methods
  M1
  ...
  Mn
end
```

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = val := !val + 1 end
    fun get() = !val end
    proc dup() = self.inc(); self.inc() end
  end
in
  decl c = new Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```

- Métodos

```
proc id() = C
fun  id() = E
```


Tradução para Core

- Construções novas podem ser implementadas por tradução nas construções já existentes na linguagem base.

```
decl Counter =
  class
    val := 0
  methods
    proc inc() = v
    fun get() = !v
    proc dup() = s
  end
in
  decl c = new Cou
    in c.inc(); c.
  end
end
```

```
decl Counter =
  fun =>
    decl val = var(0) in
    declrec self =
    [
      inc = proc => val := !val + 1 end,
      get = fun => !val end
      dup = proc => self.inc(); self.inc() end
    ] in self end
  end
in
  decl c = Counter()
    in c.inc(); c.dup(); print(c.get())
  end
end
```