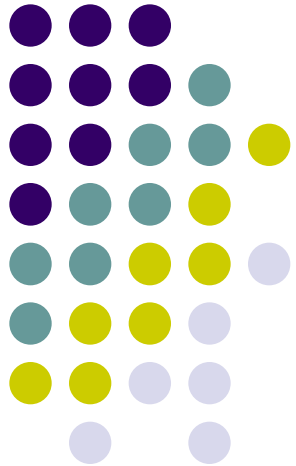
Modelos de Linguagens de Programação

– Aula 04 –

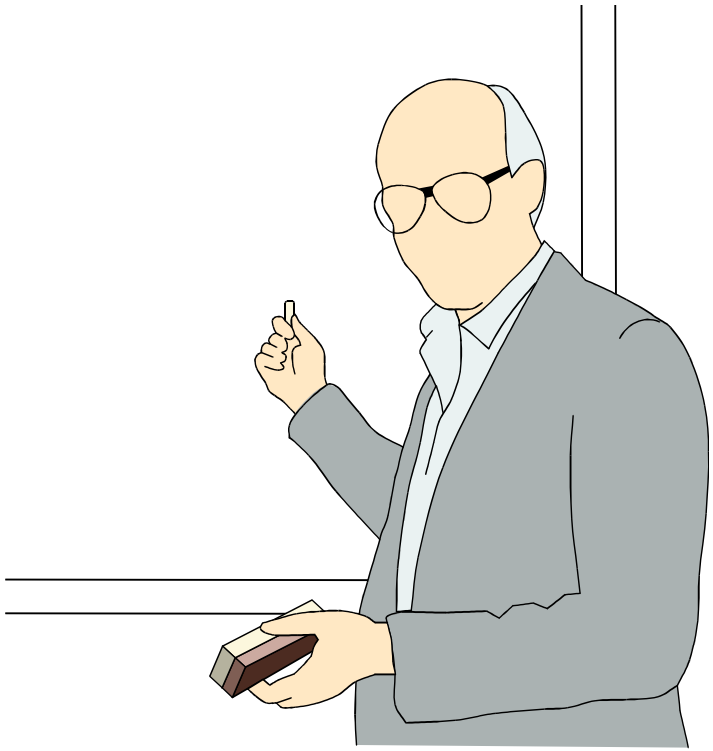
Laboratório de Programação Funcional

#1/3

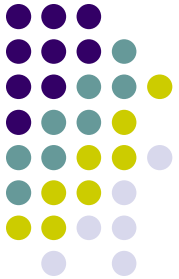


Tópicos

- Conceitos de Programação Funcional: revisão e fechamento
- Introdução à linguagem ML
- Prática de ML

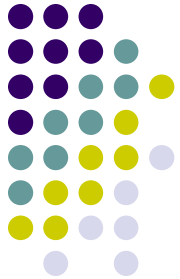


Revisão/fechamento



- Notação e cálculo lambda:
  - Abstração lambda: λx.x*x*x
  - Aplicação: (λx.x*x*x)(3) #resulta em 27

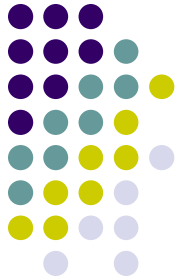
Revisão/fechamento



- Elemento de 1ª ordem:
  - São os elementos manipulados pelos comandos e operadores da linguagem (e.g., tipos primitivos)
  - Em LP funcionais, funções também são elementos de 1ª ordem

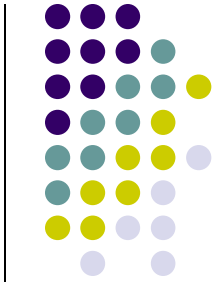
- Função de ordem superior (*high order function*):
  - Manipulam/operam sobre outras funções (aceitam outras funções como argumentos e/ou retornam outras funções como resultado)

Revisão/fechamento



- Formas funcionais (funções de ordem superior):
  - Dado que:
    - f(x) ≡ x + 2
    - g(x) ≡ x * x
    - h(x) ≡ 3 * x
  - Composição: h ≡ f • g ➔ h(x) ≡ f(g(x))
  - Construção: [f,g,h](4) ➔ (6,8,12)
  - *Apply-to-all*: α(f,(2,3,4)) ➔ (4,5,6)

Revisão/fechamento

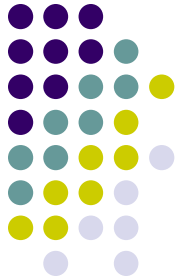


Estratégias de avaliação de expressões:

- Avaliação ansiosa (*eager evaluation*) (Tucker e Noolan (2008) chamam de avaliação rápida)

- Avaliação preguiçosa ou tardia/adiada (*lazy/delayed evaluation*) (Tucker e Noolan (2008) chamam de avaliação lenta)

Revisão/fechamento



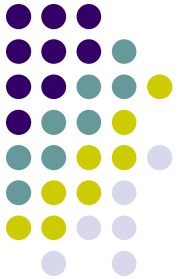
Estratégias de avaliação de expressões:

- **Considere o exemplo (imperativo):**

```
x = 5 + 3 * (1 + 5 ^ 2);
print x;
print x + 2;
```

  - **Avaliação ansiosa (*eager evaluation*)**
    - comum, principalmente em LPs imperativas
    - os parâmetros são avaliados quando a variável é definida (no caso de funções, quando elas forem chamadas)
    - no exemplo, *x* é calculado uma única vez e o resultado armazenado em memória

Revisão/fechamento



Estratégias de avaliação de expressões:

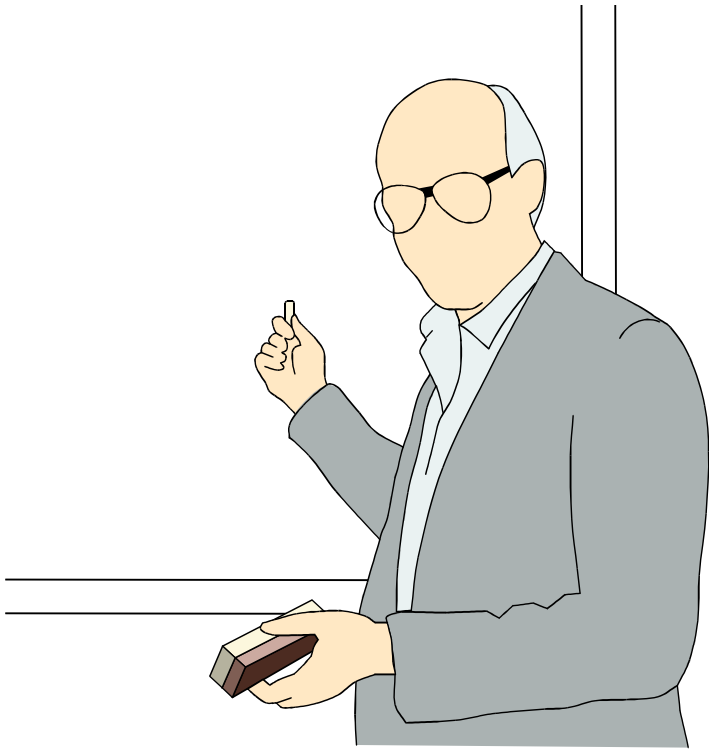
- **Considere o exemplo (imperativo):**

```
x = 5 + 3 * (1 + 5 ^ 2);
print x;
print x + 2;
```

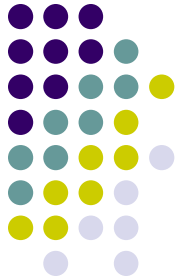
  - **Avaliação tardia (*lazy/delayed evaluation*):**
    - nenhuma subexpressão é avaliada até que seu valor seja reconhecido como necessário
    - no exemplo, *x* só é calculado quando usado no print
      - a expressão ocupa memória até esse momento!
      - *x* é recalculado no segundo uso!
    - pode ser utilizada pelas linguagens funcionais

Tópicos

- Conceitos de Programação Funcional: revisão e fechamento
- Introdução à linguagem ML
- Prática de ML

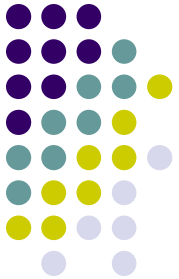


Introdução à ML



- ML = Meta Language
- Linguagem funcional de propósito geral, mas impura, pois permite efeitos colaterais e programação imperativa
- Desenvolvida por Robin Milner et al. na Universidade de Edimburgo, na década de 1970
- Padronizada como SML em 1990, revisada em 1997: SML'97
- É uma família com muitas linguagens, sendo seus principais dialetos: SML (standard ML) e CAML
- Influenciou: Haskell, Cyclone, Nemerle e F# (.NET)

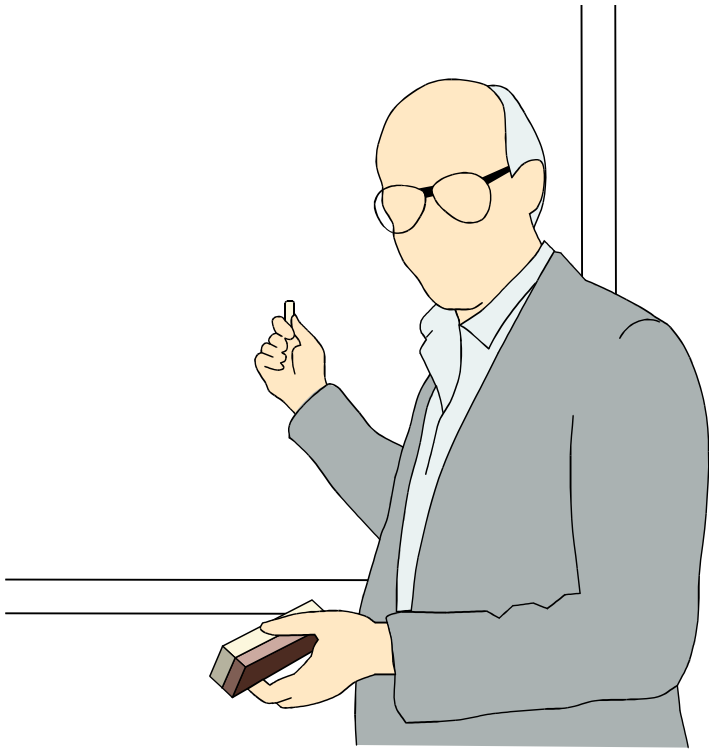
Características da ML



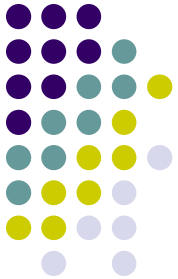
- Não usa a sintaxe funcional com parênteses
- Tem chamada por valor
- Possui inferência de tipos automática
- *Garbage collector*
- Oferece polimorfismo paramétrico
- Possui tipagem estática
- Trabalha com tipos de dados algébricos
- Utiliza *pattern matching* (casamento de padrões)
- Oferece tratamento de exceções
- Trabalha com avaliação ansiosa (mas pode simular *avaliação tardia* e listas infinitas com o uso de funções anônimas)

Tópicos

- Conceitos de Programação Funcional: revisão e fechamento
- Introdução à linguagem ML
- Prática de ML



“Hello World” funcional



- Função Fatorial (recursiva)
- Em ML fica muito similar à sintaxe matemática:

```
fun fac (0 : int) : int = 1
  | fac (n : int) : int = n * fac (n-1)
```

- As anotações de tipo podem ser omitidas (devido à inferência de tipo):

```
fun fac 0 = 1
  | fac n = n * fac(n-1)
```

Tipos primitivos em ML

- **Tipos simples**
  - **Teste os seguintes comandos no interpretador:**

    - `5;` [enter]
    - `2.0;` [enter]
    - `true;` [enter]
    - `#"B";` [enter]
    - `"Leonardo";` [enter]

  - **Quais são os tipos primitivos?**

Operadores

- **Teste os seguintes comandos no interpretador :**
  - `5 + 3 div 2;` [enter]
  - `5.0 / 2.0;` [enter]
  - `2 * 3;` [enter]
  - `3 > 2;` [enter]
  - `6 <= 10;` [enter]
  - `5 <> 6;` [enter]
  - `2 = 2;` [enter]
- Quais são os operadores aritméticos? E os relacionais?
- A linguagem é ortogonal? Até que ponto? Comente, com exemplos.

Expressões

- **Explique o que acontece em cada comando:**
  - `3.0 + 2.0 = 3 + 2;`
  - `(3.0 + 2.0) = real (3+2);`
  - `round(2.2);`
  - `val r1 = true orelse false;`
  - `val r2 = true andalso true;`
  - `val y=(10, "gato");`
  - `val (prim, seg)=y;`
  - `a;`
  - `5 - 9;`
- **Responda:**
  - A linguagem é fortemente tipada? Há conversão implícita/explícita?
  - O operador "=" possui semântica diferenciada, dependente do contexto?

Arquivos e bibliotecas

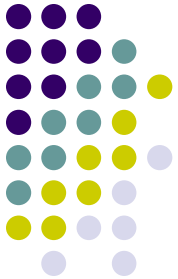
- **Carregando arquivos:**

```
- use "c:\\temp\\arquivo.ml";
```

- **Carregando bibliotecas:**

```
- load "Math";

- Math.pi; (* constante *)
- Math.sqrt(2.0); (* função predefinida *)
- help "Math"; (* lista funções da biblioteca *)
- help "Math.pow"; (* explica função *)
```

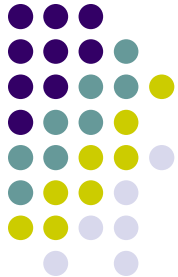


Associações (*bindings*)

- Associações de valores a nomes (comando val)
  - `val x = 10;` [enter]
  - `val e = Math.e;`
  - `x;` [enter]
  - `e;` [enter]
  - `val r = true orelse false;` [enter]
  - `val cc = bb andalso true;` [enter]

OBS:
- A precedência de operadores lógicos é igual a de linguagem C
- orelse e andalso são avaliadas em curto-circuito!

Tipos compostos



- **Listas**: elementos **entre [ ]**, de mesmo tipo.
  Exemplo: `[1, 2, 3]`

- **Tuplas**: elementos **entre ( ),** de tipos diferentes, onde a ordem importa.
  Exemplo: `(23, "Maria Silva", false, (223434, 344342))`

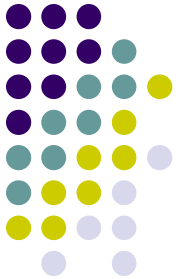
- **Registros**: elementos **nomeados**, **entre { },** de tipos diferentes, acessados em qualquer ordem.
  Exemplo: `{nome = "Lucas", idade = 23, tel = "9999999"};`

Acessando elementos...

```
- val tupla = (10,"gato"); [enter]
> val y = (10, "gato") : int * string
- val (prim, seg) = tupla; [enter]
> val prim = 10 : int
  val seg = "gato" : string
- val a1 = { nome = “Lucas”, idade = 23, sexo = #”M”};
- val i = #idade a1
> val i = 23 : int
- val e2 = #2 (26, “Laura”, #“F”);
> val e2 = “Laura” : String
```

Listas

- Teste as seguintes expressões, indicando se funcionam ou não e por que (em caso negativo):
  - `val l1 = 10::20::nil;`
  - `val l2 = l1@[30];`
  - `val l3 = l1@30;`
  - `val l4 = 30@l1;`
  - `val l5 = 50::l2;`
  - `val l6 = [50]::l2;`
  - `val l7 = l1::50;`
  - `val l8 = l1::[50];`
  - `val l9 = [l1]::[50];`



Acessando elementos...

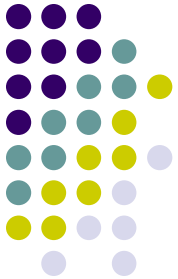
```
- val h::t = [1,2,3];   (* :: separa o primeiro elemento e coloca em h *)
> val h = 1 : int
  val t = [2,3] : int list

- val h = hd([1,2,3]);   (*comando head pega a 'cabeça' da lista *)
> val h = 1 : int

- val h = tl([1,2,3]);   (*comando tail pega a 'cauda' da lista *)
> val h = [2, 3] : int list

- val h1::h2::t= [50, 10, 20, 30];
> val h1 = 50 : int
  val h2 = 10 : int
  val t = [20, 30] : int list
```

Funções: definição



- **Sintaxe**:
  - `fun <nome> <argumentos> = <expressão>;`
- **Exemplos**:
  - `fun inc n = n + 1;`
  - `fun quadrado x = x * x;`

  - ```
    fun fat 0 = 1                    (* critério parada *)
    |   fat n = n * fat (n-1);           (*recursão *)
    ```
    ```

Funções: definição

- Há diferentes (formas de) assinaturas

```
- fun maior(a,b)=if a>b then a else b;

- maior(3,4);
> val it = 4 : int
```

Forma de uso com parênteses

```
- fun max a b = if a>b then a else b;
- max 3 4;
> val it = 4 : int
```

Forma de uso sem parênteses

```
- max (maior(3,4)) 5; (*combinando as duas*)
```