Linguagens de programação funcional

Linguagens de Programação

Marco A L Barbosa

Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.

http://github.com/malbarbo/na-lp-copl

Conteúdo

Introdução

Funções matemáticas

Fundamentos das linguagens de programação funcionais

A primeira linguagem de programação funcional: Lisp

Introdução ao Scheme

Common Lisp

ML

Haskell

Aplicações das linguagens funcionais

Comparação entre as linguagens funcionais e imperativas

Referências

Introdução

Introdução

- O projeto das linguagens imperativas é baseado na arquitetura de von Neumann
  - Eficiência é a preocupação primária, ao invés da adequação da linguagem para o desenvolvimento de software
  - As linguagens imperativas podem ser vistas como uma sucessão de melhorias ao modelo básico (Fortran 1)

Introdução

- O projeto das linguagens imperativas é baseado na arquitetura de von Neumann
  - Eficiência é a preocupação primária, ao invés da adequação da linguagem para o desenvolvimento de software
  - As linguagens imperativas podem ser vistas como uma sucessão de melhorias ao modelo básico (Fortran 1)
- O projeto das linguagens funcionais é baseado em funções matemáticas
  - Uma base teórica sólida
  - Sem preocupação com a arquitetura da máquina

Introdução

- O projeto das linguagens imperativas é baseado na arquitetura de von Neumann
  - Eficiência é a preocupação primária, ao invés da adequação da linguagem para o desenvolvimento de software
  - As linguagens imperativas podem ser vistas como uma sucessão de melhorias ao modelo básico (Fortran 1)
- O projeto das linguagens funcionais é baseado em funções matemáticas
  - Uma base teórica sólida
  - Sem preocupação com a arquitetura da máquina
- O interesse pelo paradigma de programação funcional cresceu após o discurso do John Backus quando ele recebeu o ACM Turing Award em 1977
  - Os programas escritos em linguagens de programação puramente funcionais são mais legíveis, mais confiáveis, e tem mais chances de estarem corretos
  - O significado das expressões são independentes do contexto

Funções matemáticas

Funções matemáticas

- Uma função matemática é um mapeamento dos membros de um conjunto, chamado de conjunto domínio, para outro, o conjunto imagem
- Note que os conjuntos podem ser o produto cartesiano de diversos outros conjuntos
- A ordem de avaliação das expressões de uma função é controlada por recursão e por expressões condicionais, não pela sequência ou pela repetição interativa (comum nas linguagens imperativas)
- Uma função matemática sempre define o mesmo valor, para os mesmos argumentos

Funções matemáticas

- Funções simples
  - Definição de função: $\text{cubo}(x) \equiv x * x * x$
  - Aplicação de função: $\text{cubo}(8)$

Funções matemáticas

- Funções simples
  - Definição de função: $\text{cubo}(x) \equiv x * x * x$
  - Aplicação de função: $\text{cubo}(8)$
- As vezes é conveniente separar a atividade de definir uma função e a atividade de nomear uma função
  - Notação lambda (Alonzo Church 1941)
  - Definição: $\lambda(x)x * x * x$
  - Aplicação: $(\lambda(x)x * x * x)(2)$

Funções matemáticas

- Formas funcionais
  - Uma função de alta ordem, ou forma funcional, é aquela que toma funções como parâmetro, produz uma função como resultado, ou ambos

Funções matemáticas

- Formas funcionais
  - Uma função de alta ordem, ou forma funcional, é aquela que toma funções como parâmetro, produz uma função como resultado, ou ambos
- Composição de funções
  - Toma duas funções como parâmetro e define uma função cujo o valor é a primeira função aplicada ao resultado da segunda
  - $h \equiv f \circ g$, $h(x) \equiv f(g(x))$

Funções matemáticas

- Formas funcionais
  - Uma função de alta ordem, ou forma funcional, é aquela que toma funções como parâmetro, produz uma função como resultado, ou ambos
- Composição de funções
  - Toma duas funções como parâmetro e define uma função cujo o valor é a primeira função aplicada ao resultado da segunda
  - $h \equiv f \circ g$, $h(x) \equiv f(g(x))$
- Aplicar a tudo (Apply to all)
  - Toma uma função como parâmetro e resulta em uma lista de valores aplicando a função dada a cada elemento da lista de parâmetros
  - Forma: α
  - $h(x) \equiv x * x$
  - $\alpha(h, (2, 3, 4))$ resulta $(4, 9, 16)$

Fundamentos das linguagens de programação funcionais

Fundamentos das linguagens de programação funcionais

- O objetivo do projeto de uma LPF é imitar ao máximo as funções matemáticas

Fundamentos das linguagens de programação funcionais

- O objetivo do projeto de uma LPF é imitar ao máximo as funções matemáticas
- O processo básico de computação em uma LPF é fundamentalmente diferente do processo em uma linguagem imperativa
  - Em uma linguagem imperativa, as operações são realizadas e os resultados são armazenados em variáveis para uso posterior

Fundamentos das linguagens de programação funcionais

- O objetivo do projeto de uma LPF é imitar ao máximo as funções matemáticas
- O processo básico de computação em uma LPF é fundamentalmente diferente do processo em uma linguagem imperativa
  - Em uma linguagem imperativa, as operações são realizadas e os resultados são armazenados em variáveis para uso posterior
  - Nas LFP, variáveis não são necessárias, assim como na matemática
  - Nas LFP, a avaliação de uma função sempre produz o mesmo resultado se os mesmos parâmetros forem dados (transparência referencial)
  - Repetição é especificada com recursão
  - Programas consistem em definições de funções e especificação de aplicações de funções
  - A execução consistem na avaliação das aplicações das funções

Fundamentos das linguagens de programação funcionais

- Uma LPF deve prover
  - Um conjunto de funções primitivas
  - Um conjunto de formas funcionais
  - Um operador de aplicação de função
  - E algumas estruturas para representar dados
- Em geral as LPF são implementadas com interpretadores, mas elas também podem ser compiladas
- As linguagens imperativas em geral oferecem suporte limitado a programação funcional
  - Não oferecem muitas formas funcionais (ex: retorno de função)
  - Permitem efeitos colaterais

A primeira linguagem de programação funcional: Lisp

A primeira linguagem de programação funcional: Lisp

- Lisp (LISt Processing) foi criada por John McCarthy em 1958
- É a segunda linguagem de programação de alto nível mais antiga ainda em uso
- Existem muitos dialetos: Common Lisp, Scheme, Clojure

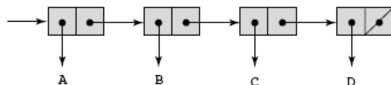
A primeira linguagem de programação funcional: Lisp

- Tipo de dados
  - Átomos: símbolos (identificadores) ou literais numéricos
  - Listas
- Listas são especificadas delimitando os seus elementos com parenteses
  - `(A B C D)`
  - `(A (B C) D (E (F G)))`
  - Em geral, as listas são armazenadas internamente como listas ligada simples

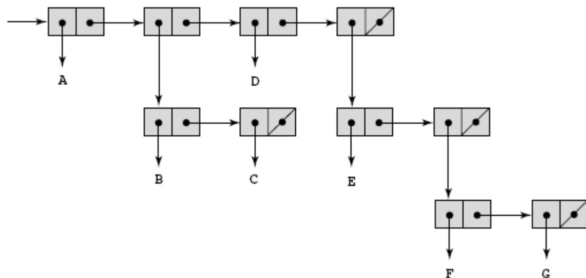
A primeira linguagem de programação funcional: Lisp

Figure 15.1

Internal representation of two LISP lists



(A B C D)



(A (B C) D (E (F G)))

A primeira linguagem de programação funcional: Lisp

- A notação lambda é usada para especificar funções e definição de funções
  - (function_name (lambda (arg1 ... argn) expression))

A primeira linguagem de programação funcional: Lisp

- A notação lambda é usada para especificar funções e definição de funções
  - (function_name (lambda (arg1 ... argn) expression))
- A aplicação de funções e os dados tem a mesma forma
  - Se a lista (A B C) é interpretada como dado ela é uma lista simples de três átomos
  - Se a lista é interpretada como aplicação de função, ela significa que a função nomeada A é aplicada aos parâmetros A e B

A primeira linguagem de programação funcional: Lisp

- A notação lambda é usada para especificar funções e definição de funções
  - (function_name (lambda (arg1 ... argn) expression))
- A aplicação de funções e os dados tem a mesma forma
  - Se a lista (A B C) é interpretada como dado ela é uma lista simples de três átomos
  - Se a lista é interpretada como aplicação de função, ela significa que a função nomeada A é aplicada aos parâmetros A e B
- Lisp foi a primeira linguagem homoicônica
- O primeiro interpretador Lisp apareceu como uma demonstração da capacidade universal de computação da notação
  - Stephen Russell e Daniel Edwards
  - Implementação da função eval
- A notação usada pelo Lisp é chamada de expressões s (symbolic expressions)

Introdução ao Scheme

Introdução ao Scheme

- Criada em meados de 1970 por Gerald Sussman e Guy Steele, projetada uma versão para ser limpa, moderna e simples de Lisp
- Usa apenas escopo estático
- Função são entidades de primeira classe

Introdução ao Scheme

- Avaliação de aplicação de funções
  - Os parâmetros são avaliados, em nenhuma ordem particular
  - Os valores dos parâmetros são substituídos no corpo da função
  - O corpo da função é avaliada
  - O valor da última expressão no corpo da função é o resultado
- Exemplo
  - `(* (- 5 3) (/ 8 2))`
  - `(* 2 (/ 8 2))`
  - `(* 2 4)`
  - 8

Introdução ao Scheme

- Funções numéricas
  - Aritmética: `+`, `-`, `*`, `/`, `abs`, `quotient`, `remainder`, `modulo`, `gcd`, `lcm`, `expt`, `sqrt`
  - Aproximação: `floor`, `ceiling`, `truncate`, `round`
  - Desigualdades: `<`, `<=`, `>`, `>=`
  - Outros: `zero?`, `negative?`, `positive?` `odd?` `even?`
  - Máximo e mínimo: `max`, `min`
  - Trigonometria: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
  - Exponencial: `exp`, `log`

Introdução ao Scheme

- Formas especiais

```
(define symbol expression)

(lambda (parameters) expression)

(define (function-name parameters) expression)

(if predicate then-expression else-expression)

(cond
    (predicate1 expression1)
    (predicate2 expression2)
    ...
    (predicaten expressionn)
    [(else expression)]
)
```

Introdução ao Scheme

- Formas especiais
  - `(quote expression)`

    ```
    > (quote a)
    a
    > (quote (a b c))
    (a b c)
    > '(+ 2 3)
    (+ 2 3)
    ```

  - `(let ((id exp)+) exp)`

    ```
    > (let ((a 10) (b 20)) (+ a b))
    30
    ```

  - `let*, letrec`

Introdução ao Scheme

- Listas
  - car: retorna o primeiro elemento de uma lista

```
> (car '(a b c))
'a
> (car '((a b) c d))
'(a b)
> (car 'a)
erro
> (car '(a))
'a
> (car '())
erro
```

Introdução ao Scheme

- Listas
  - cdr: retorna a lista sem o primeiro elemento

```
> (cdr '(a b c))
'(b c)
> (cdr '((a b) c d))
'(c d)
> (cdr 'a)
erro
> (cdr '(a))
'()
> (cr '())
erro
```

- Listas
  - `cons`: insere o primeiro parâmetro como car do segundo parâmetro

    ```
    > (cons 'a '())
    '(a)
    > (cons 'a '(b c))
    '(a b c)
    > (cons '() '(a b))
    '(() a b)
    > (cons '(a b) '(c d))
    '((a b) c d)
    > (cons 'a 'b)
    '(a . b)
    ```

  - `list`: constrói uma lista a partir dos parâmetros

    ```
    > (list 'a 'b 'c)
    '(a b c)
    ```

Introdução ao Scheme

- Funções predicados para listas
  - `eq?`: recebe dois parâmetros e faz uma comparação rasa, isto é, verifica se os dois parâmetros referem-se ao mesmo objeto
  - `eqv?`: semelhante ao `eq?`, mas para números e caracteres a comparação é pelo conteúdo, não pela referência
  - `equal?`: para os tipos definidos pelo Racket (lista, string, etc), faz comparação profunda, ou seja, verifica recursivamente se o conteúdo é o mesmo
  - `list?`: retorna `#t` se o parâmetro é uma lista
  - `null?`: retorna `#t` se o parâmetro é uma lista vazia

Introdução ao Scheme

- Recursão em cauda
  - Uma **função recursiva em cauda** é aquela em que a chamada recursiva é a última operação na função
  - Este tipo de função pode ser traduzida pelo compilador em uma iteração, resultando em uma execução mais rápida
  - As vezes é interessante transformar uma função recursiva em uma função recursiva em cauda

```
(define (fat-helper n partial)
  (if (= n 0)
      partial
      (fat-helper (- n 1) (* n partial))))

(define (fat n)
  (fat-helper n 1))
```

Introdução ao Scheme

- Exemplos
  - Testar se um elemento é membro de uma lista
  - Calcular o tamanho de uma lista
  - Calcular a soma dos elementos de uma lista
  - Calcular o produto dos elementos de uma lista
  - Reversão de lista
  - Testar se duas listas são iguais
  - Concatenação de duas listas
  - Interseção de duas listas
  - Ordenação (quicksort)
  - Etc

Introdução ao Scheme

- Formas funcionais
  - `foldl`, `foldr`
  - `map`
  - `filter`
- Outras funções
  - `eval`
  - `apply`

Common Lisp

Common Lisp

- Foi criado para combinar características de diversos dialetos de Lisp
- É uma linguagem grande e complexa (oposto do Scheme)
- Características
  - Permite escopo estático e dinâmico
  - Muito tipos de dados e estruturas
  - Sistema poderoso de entrada e saída
  - Pacotes (para modularização)
- Common Lisp foi criado para ser uma linguagem comercial
- Scheme é mais usado para o ensino

ML

ML

- Criada por Robert Miler (e outros) no início dos anos 70
- Fortemente tipada
- Sintaxe semelhante as linguagens imperativas
- Inclui tratamento de exceção, módulos (para criar tipos abstratos de dados) e listas

ML

- Declaração de função

```
fun nome_da_função(parâmetros) = corpo_da_função;

fun square(x : int) = x * x

fun square(x : int) : int = x * x

fun square(x) = x * x
```

- Funções que usam aritmética não podem ser polimórficas
- Funções que usam apenas operações de listas, $=$, $<>$, e operações de tuplas, podem ser polimórficas
- Funções definidas pelo usuário não podem ser sobrecarregadas

ML

- Controle de fluxo

```
if expressão then expressão_then else expressão_else
```

ML

- Controle de fluxo

```
if expressão then expressão_then else expressão_else
```

- Exemplo

```
fun fact(n : int): int = if n = 0 then 1
                         else n * fact(n - 1);
```

ML

- Controle de fluxo

```
if expressão then expressão_then else expressão_else
```

- Exemplo

```
fun fact(n : int): int = if n = 0 then 1
                         else n * fact(n - 1);
```

- Múltiplas definições de uma função podem ser escritas usando casamento de padrão nos parâmetros

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

ML

- Listas
  - Listas literais são especificadas com colchetes: `[5, 7, 9]`
  - Lista vazia, `[]` ou `nill`
  - Todos os elementos da lista precisam ser do mesmo tipo
  - A operação `cons` é o operador binário `::`
  - `3 :: [5, 7, 9]` resulta em `[3, 5, 7, 9]`
  - `car` é a função unária `hd` (head)
  - `cdr` é a função unária `tl` (tail)
  - As funções `hd` e `tl` são menos usados do que em Lisp, porque as funções podem ser definidas com padrões

ML

- Listas
  - Listas literais são especificadas com colchetes: `[5, 7, 9]`
  - Lista vazia, `[]` ou `nill`
  - Todos os elementos da lista precisam ser do mesmo tipo
  - A operação cons é o operador binário `::`
  - `3 :: [5, 7, 9]` resulta em `[3, 5, 7, 9]`
  - `car` é a função unária `hd` (head)
  - `cdr` é a função unária `tl` (tail)
  - As funções `hd` e `tl` são menos usados do que em Lisp, porque as funções podem ser definidas com padrões
  - Exemplos

```
fun lenght([]) = 0
|   lenght(h :: t) = 1 + lenght(t);

fun append([], list2) = list2
|   append(h :: t, list2) = h :: append(t, list2);
```

ML

- Declaração de nomes
  - A instrução `val` vincula um nome a um valor
  - `val distance = time * speed;`
  - Usado na cláusula `let`

```
let
  val pi = 3.14159
in
  pi * radius * radius
end;
```

ML

- ML tem tido um impacto significativo na evolução das linguagens de programação
- Foi uma das linguagens mais estudas em pesquisas
- Originou diversas outras linguagens
  - Ocaml
  - F#
- Influenciou
  - Miranda
  - Haskell
  - Scala

Haskell

Haskell

- Similar a ML: sintaxe parecida, escopo estático, fortemente tipada, e usa o mesmo método de inferência de tipo
- Duas características que a diferenciam de ML
  - As funções Haskell podem ser polimórficas
  - Usa avaliação não estrita

Haskell

- Exemplos de funções

```
fact 0 = 1
fact n = n * fact (n - 1)

fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Haskell

- Guardas podem ser adicionadas as funções (expressões condicionais)

```
fact n
  | n == 0 = 1
  | n > 0  = n * fact(n - 1)

sub n
  | n < 10    = 0
  | n > 100   = 2
  | otherwise = 1
```

Haskell

- Guardas podem ser adicionadas as funções (expressões condicionais)

```
fact n
  | n == 0 = 1
  | n > 0  = n * fact(n - 1)

sub n
  | n < 10    = 0
  | n > 100   = 2
  | otherwise = 1
```

- Exemplo de função polimórfica

```
square x = x * x
```

Haskell

- Listas
  - Especificadas com colchetes
  - Concatenação ++
  - Cons infixo :
  - Série aritmética ..

Haskell

- Listas
  - Especificadas com colchetes
  - Concatenação ++
  - Cons infixo :
  - Série aritmética ..
  - Exemplos

```
# 5:[2, 7, 9]
[5, 2, 7, 9]

# [1, 3..11]
[1, 3, 5, 7, 9, 11]

# [1, 3, 5] ++ [2, 4, 6]
[1, 3, 5, 2, 4, 6]

product [] = 1
product (a:x) = a * product x

fact n = product [1 .. n]
```

Haskell

- List comprehentions
  - Descreve a criação de listas a partir de outras listas
  - Sintaxe `[body | qualifiers]`

Haskell

- List comprehentions
  - Descreve a criação de listas a partir de outras listas
  - Sintaxe [body | qualifiers]
  - Exemplos

```
[n * n * n | n <- [1..50]]

factors n = [i | i <- [1..n `div` 2], n `mod` i==0]
```

Haskell

- List comprehentions
  - Descreve a criação de listas a partir de outras listas
  - Sintaxe [body | qualifiers]
  - Exemplos

```
[n * n * n | n <- [1..50]]

factors n = [i | i <- [1..n `div` 2], n `mod` i==0]

sort [] = []
sort (h:t) = sort [b | b <- t, b <= h]
             ++ [h] ++
             sort [b | b <- t, b >  h]
```

Haskell

- Avaliação atrasada
  - Uma linguagem é estrita se ela requer que todos os parâmetro reais sejam completamente avaliados
  - Uma linguagem é não estrita se não tem este requisito
  - Uma forma de avaliação utilizada pelas linguagens não estritas é a avaliação atrasada
  - Na avaliação atrasada, uma expressão só é avaliada se e quando seu valor for necessário
  - Dificulta pensar sobre a quantidade de recursos usados pelo programa
  - Exemplos

Haskell

- Avaliação atrasada
  - Uma linguagem é estrita se ela requer que todos os parâmetro reais sejam completamente avaliados
  - Uma linguagem é não estrita se não tem este requisito
  - Uma forma de avaliação utilizada pelas linguagens não estritas é a avaliação atrasada
  - Na avaliação atrasada, uma expressão só é avaliada se e quando seu valor for necessário
  - Dificulta pensar sobre a quantidade de recursos usados pelo programa
  - Exemplos

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <- positives]
```

Haskell

- Avaliação atrasada
  - Uma linguagem é estrita se ela requer que todos os parâmetro reais sejam completamente avaliados
  - Uma linguagem é não estrita se não tem este requisito
  - Uma forma de avaliação utilizada pelas linguagens não estritas é a avaliação atrasada
  - Na avaliação atrasada, uma expressão só é avaliada se e quando seu valor for necessário
  - Dificulta pensar sobre a quantidade de recursos usados pelo programa
  - Exemplos

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <- positives]

member squares 16
```

  - Como implementar a a função `member`?

Haskell

- Definição da função member

```haskell
member [] n = False
member (h:xs) n = (h == n) || member xs n
```

Haskell

- Definição da função `member`

```
member [] n = False
member (h:xs) n = (h == n) || member xs n
```

- Problema: esta definição pode gerar um laço infinito!

Haskell

- Definição da função `member`

```
member [] n = False
member (h:xs) n = (h == n) || member xs n
```

- Problema: esta definição pode gerar um laço infinito!

```
member2 [] n = False
member2 (h:xs) n
  | h < n     = member2 xs n
  | h == n    = True
  | otherwise = False
```

Haskell

- Definição da função member

```
member [] n = False
member (h:xs) n = (h == n) || member xs n
```

- Problema: esta definição pode gerar um laço infinito!

```
member2 [] n = False
member2 (h:xs) n
  | h < n     = member2 xs n
  | h == n    = True
  | otherwise = False
```

- Esta versão funciona corretamente

Aplicações das linguagens funcionais

Aplicações das linguagens funcionais

- Lisp é usado em programas de inteligência artifical e como uma linguagem de extensão
- Lisp: emacs, macsyma (maxima), AutoCAD
- Scheme é utilizado para o ensino de programação funcional
- Haskell: darcs, pugs
- Outras linguagens: Clojure, Erlang, Scala

Comparação entre as linguagens funcionais e imperativas

Comparação entre as linguagens funcionais e imperativas

- Linguagens imperativas
  - Execução eficiente (nem sempre)
  - Semântica complexa
  - Sintaxe complexa
  - Concorrência é projetada pelo programador
- Linguagens funcionais
  - Execução ineficiente (nem sempre)
  - Semântica simples
  - Sintaxe simples
  - Os programas podem automaticamente serem feitos concorrentes

Comparação entre as linguagens funcionais e imperativas

```
int sum_cubes(int n) {
  int sum = 0;
  for (int i = 1; i <= i; i++) {
    sum += i * i * i
  }
  return sum
}
```

```
sum_cubes n = sum (map (^3) [1..n])
```

Referências

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 15.