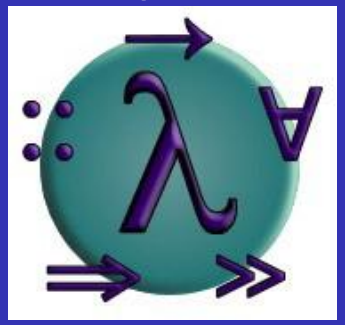
Programação Funcional



Capítulo 1

Introdução

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2014.1

Tópicos

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Paradigmas de programação

- Um **paradigma de programação** fornece e determina a visão que o programador possui sobre a estruturação e a execução do programa.
- Por exemplo:
  - Em programação orientada a objetos, programadores podem abstrair um programa como uma *coleção de objetos* que interagem entre si.
  - Em programação lógica os programadores abstraem o programa como um *conjunto de predicados* que estabelecem relações entre objetos, e uma *meta* a ser provada usando os predicados.
- Diferentes linguagens de programação propõem diferentes paradigmas de programação.
- Algumas linguagens foram desenvolvidas para suportar um paradigma específico.
- Por exemplo:
  - Smalltalk, Eiffel e Java suportam o paradigma de orientação a objetos.
  - Haskell e Clean suportam o paradigma funcional.
  - OCaml, LISP, Scala, Perl, Python e C++ suportam múltiplos paradigmas.

Técnicas e paradigmas de programação

- Geralmente os paradigmas de programação são diferenciados pelas técnicas de programação que **permitem** ou **proíbem**.
- Por exemplo, a programação estruturada não permite o uso de *goto*.
- Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais.
- Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de correção de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.
- O relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma.

Técnicas e paradigmas de programação

- Geralmente os paradigmas de programação são diferenciados pelas técnicas de programação que **permitem** ou **proíbem**.
- Por exemplo, a programação estruturada não permite o uso de *goto*.
- Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais.
- Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de correção de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.
- O relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma.

Técnicas e paradigmas de programação

- Geralmente os paradigmas de programação são diferenciados pelas técnicas de programação que **permitem** ou **proíbem**.
- Por exemplo, a programação estruturada não permite o uso de *goto*.
- Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais.
- Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de correção de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.
- O relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma.

Técnicas e paradigmas de programação

- Geralmente os paradigmas de programação são diferenciados pelas técnicas de programação que **permitem** ou **proíbem**.
- Por exemplo, a programação estruturada não permite o uso de *goto*.
- Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais.
- Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de correção de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.
- O relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma.

Técnicas e paradigmas de programação

- Geralmente os paradigmas de programação são diferenciados pelas técnicas de programação que **permitem** ou **proíbem**.
- Por exemplo, a programação estruturada não permite o uso de *goto*.
- Esse é um dos motivos pelo qual novos paradigmas são considerados mais rígidos que estilos tradicionais.
- Apesar disso, evitar certos tipos de técnicas pode facilitar a prova de correção de um sistema, podendo até mesmo facilitar o desenvolvimento de algoritmos.
- O relacionamento entre paradigmas de programação e linguagens de programação pode ser complexo pelo fato de linguagens de programação poderem suportar mais de um paradigma.

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Categorias: programação imperativa e declarativa

Programação imperativa

- Descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa, enfatizando *como* resolver um problema.
- Muito parecido com o comportamento imperativo das linguagens naturais que expressam ordens, programas imperativos são uma sequência de comandos para o computador executar.
- O nome do paradigma, imperativo, está ligado ao tempo verbal imperativo, onde o programador diz ao computador: faça isso, depois isso, depois aquilo...
- Exemplos: paradigmas procedimental (C, Pascal, etc) e orientado a objetos (Smalltalk, Java, etc).

Programação declarativa

- Descreve o que o programa faz e não *como* seus procedimentos funcionam.
- Ênfase nos resultados, no que se deseja obter.
- Exemplos: paradigmas funcional (Haskell, OCaml, LISP, etc) e lógico (Prolog, etc.).

Programação funcional

- **Programação funcional** é um paradigma de programação que descreve a computação como uma expressão a ser avaliada, sendo a definição e a aplicação de funções a principal forma de estruturá-la.

Exemplo: quick sort em C

```c
// To sort array a[] of size n: qsort(a,0,n-1)

void qsort(int a[], int lo, int hi) {
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    a[hi] = a[l];
    a[l] = p;

    qsort(a, lo, l-1);
    qsort(a, l+1, hi);
  }
}
```

Exemplo: quick sort em Haskell

```
qsort []     = []
qsort (x:xs) = qsort (filter (<x) xs) ++
               [x] ++
               qsort (filter (>x) xs)
```

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

A Crise do Software

- *Linguagens declarativas* (incluindo linguagens funcionais):
  - permitem que programas sejam escritos de forma clara, concisa, e com um alto nível de abstração;
  - suportam componentes de software reutilizáveis;
  - incentivam o uso de verificação formal;
  - permitem prototipagem rápida;
  - fornecem poderosas ferramentas de resolução de problemas.
- Estas características podem ser úteis na abordagem de dificuldades encontradas no desenvolvimento de software:
  - o tamanho e a complexidade dos programas de computador modernos
  - o tempo e o custo de desenvolvimento do programas
  - confiança de que os programas já concluídos funcionam corretamente
- As linguagens funcionais oferecem um quadro particularmente elegante para abordar estes objetivos.

Tópicos

Algumas características de Haskell

- Programas concisos
- Sistema de tipos poderoso
- Compreensões de lista
- Funções recursivas
- Funções de ordem superior
- Computações monádicas
- Avaliação *lazy*
- Maior facilidade de raciocínio sobre programas

Tópicos

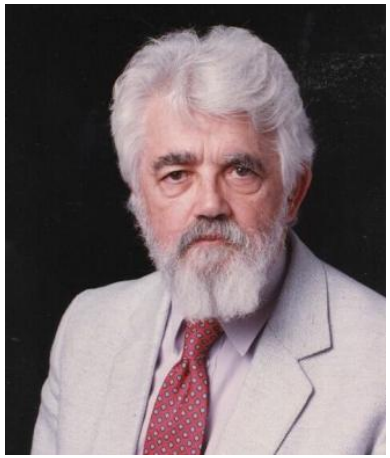
Antecedentes históricos

- Década de 1930:

Alonzo Church desenvolve o **cálculo lambda**, uma teoria de funções simples, mas poderosa.

Antecedentes históricos (cont.)

- Década de 1950:



John McCarthy desenvolve **Lisp**, a primeira linguagem funcional, com algumas influências do cálculo lambda, mas mantendo as atribuições de variáveis.

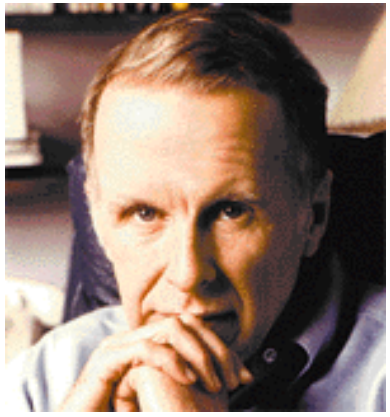
Antecedentes históricos (cont.)

- Década de 1960:

Peter Landin desenvolve **ISWIM**, a primeira linguagem funcional pura, baseada fortemente no cálculo lambda, sem atribuições.

Antecedentes históricos (cont.)

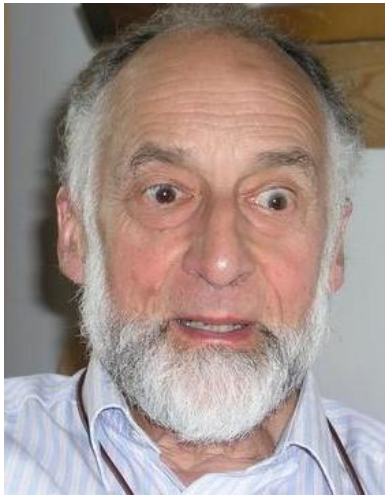
- Década de 1970:



John Backus desenvolve **FP**, uma linguagem funcional que enfatiza funções de ordem superior e raciocínio sobre programas.

Antecedentes históricos (cont.)

- Década de 1970:



Robin Milner e outros desenvolvem **ML**, a primeira linguagem funcional moderna, que introduziu a inferência de tipos e tipos polimórficos.

Antecedentes históricos (cont.)

- Décadas de 1970 e 1980:

David Turner desenvolve uma série de linguagens funcionais com *avaliação lazy*, culminando com o sistema **Miranda**.

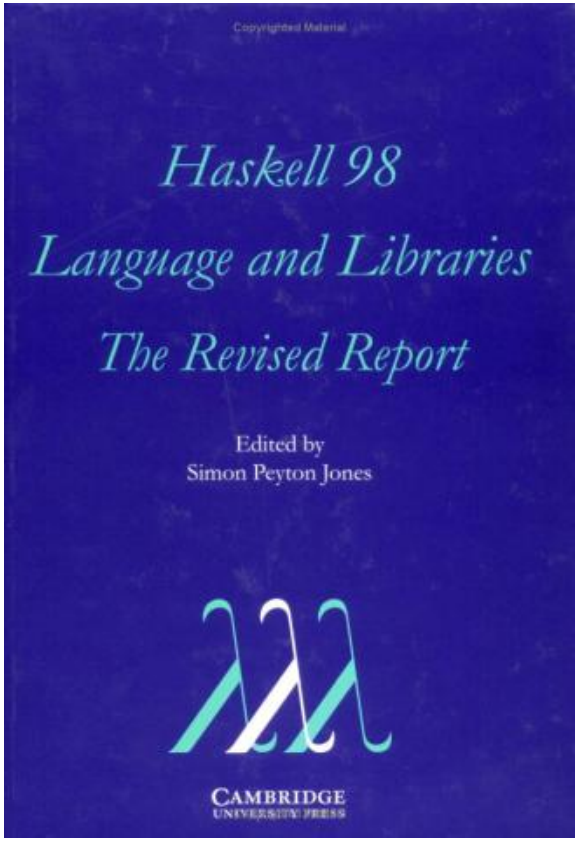
Antecedentes históricos (cont.)

- 1987:

Um comitê internacional de pesquisadores inicia o desenvolvimento de **Haskell**, uma linguagem funcional *lazy* padrão.

Antecedentes históricos (cont.)

- 2003:



O comitê publica o relatório **Haskell 98**, a definição de uma versão estável da linguagem Haskell.

Antecedentes históricos (cont.)

- 2009:

O comitê publica o relatório **Haskell 2010**, uma revisão da definição da linguagem Haskell.

Tópicos

Quem usa Haskell?

Exemplos de empresas que usam Haskell:

- **AT&T** automatização de processamento de formulários
- **Bank of America Merril Lynch** transformação de dados
- **Bump** servidores baseados em Haskell
- **Facebook** manipulação da base de código PHP
- **Google** infra-estrutura interna de TI
- **MITRE** análise de protocolos de criptografia
- **NVIDIA** ferramentas usadas internamente
- **Qualcomm, Inc** geração de interfaces de programação para Lua

Para maiores detalhes veja Haskell in industry:
`http://www.haskell.org/haskellwiki/Haskell_in_industry`.

Tópicos

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Função

- **Função** é uma relação que associa cada valor de um domínio (argumento) a um único valor de um contra-domínio (resultado).
- Em geral o mapeamento é estabelecido por uma lei de associação, expressa por um algoritmo.
- Em Haskell uma função é definida usando uma ou mais **equações**.
- Uma **equação** especifica:
  - o nome da função,
  - os parâmetros formais, que representam os argumentos, e
  - o corpo, que especifica o resultado em termos dos parâmetros formais.
- **Exemplo**:
  Uma função chamada `dobro` que recebe um número x como argumento, e produz o resultado $x + x$:

```
dobro x = x + x
```

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Aplicação de função

- Quando uma função é **aplicada** aos argumentos atuais, o resultado é obtido pela substituição desses argumentos no corpo da função no lugar dos nomes dos argumentos.
- Este processo pode produzir imediatamente um resultado que não pode ser mais simplificado, como por exemplo um número.
- Mais comumente, no entanto, a substituição produz uma expressão contendo outras aplicações de função, que devem, então, ser processadas da mesma maneira para produzir o resultado final.
- Em geral, a ordem na qual funções são aplicadas em um cálculo
  - não afeta o valor do resultado final, mas
  - pode afetar o número de passos necessários, e
  - pode determinar se o processo de cálculo termina ou não termina.

Exemplos: aplicação de função

- **Exemplo**:
  Aplicação da função `dobro` no argumento 3

```
dobro 3
= { aplicando dobro }
3 + 3
= { aplicando + }
6
```

Exemplos: aplicação de função (cont.)

- **Exemplo**:
  Aplicação aninhada de `dobro`

```
dobro (dobro 2)
= { aplicando dobro interno }
dobro (2 + 2)
= { aplicando + }
dobro 4
= { aplicando dobro }
4 + 4
= { aplicando + }
8
```

Exemplos: aplicação de função (cont.)

- **Exemplo**:
  Aplicação aninhada de `dobro`, calculada de outra maneira

```
dobro (dobro 2)
= { aplicando dobro externo }
dobro 2 + dobro 2
= { aplicando o primeiro dobro }
(2 + 2) + dobro 2
= { aplicando o primeiro + }
4 + dobro 2
= { aplicando dobro }
4 + (2 + 2)
= { aplicando o segundo + }
4 + 4
= { aplicando + }
8
```

O que é uma linguagem funcional?

As opiniões divergem, e é difícil dar uma definição precisa, mas de um modo geral:

- **Programação funcional** é um estilo de programação em que o método básico de computação é a aplicação de funções a argumentos.
- Uma **linguagem funcional** é aquela que apoia e incentiva o estilo funcional.

O que é uma linguagem funcional? (cont.)

Exemplo:
Somando os inteiros 1 a 10 em C:

```
int total = 0;
for (int i = 1; i <= 10; ++i)
   total = total + i;
```

- O método de cálculo é atribuição de variável.
- Em geral, linguagens de programação em que o método básico de computação consiste em mudar os valores armazenados em variáveis são chamadas de **linguagens imperativas**, pois os programas nestas linguagens são construídos a partir de instruções imperativas que especificam precisamente como o cálculo deve ser realizado.

O que é uma linguagem funcional? (cont.)

Exemplo:

Somando os inteiros 1 a 10 em Haskell:

```
sum [1..10]
```

- O método de cálculo é aplicação de função.

Tópicos

Experimentando Haskell

```haskell
sum :: Num a => [a] -> a

sum []     = 0
sum (x:xs) = x + sum xs
```

```haskell
sum [1, 2, 3]
= { aplicando sum }
1 + sum [2, 3]
= { aplicando sum }
1 + (2 + sum [3])
= { aplicando sum }
1 + (2 + (3 + sum []))
= { aplicando sum }
1 + (2 + (3 + 0))
= { aplicando + }
6
```

Experimentando Haskell (cont.)

```haskell
qsort []     = []
qsort (x:xs) = qsort ys ++ [x] ++ qsort zs
               where
                 ys = [a | a <- xs, a <= x]
                 zs = [b | b <- xs, b > x]
```

?

Experimentando Haskell (cont.)

```
qsort [x]
= { aplicando qsort }
qsort [] ++ [x] ++ qsort []
= { aplicando qsort }
[] ++ [x] ++ []
= { aplicando ++ }
[x]
```

Experimentando Haskell (cont.)

```
qsort [3, 5, 1, 4, 2]
= { aplicando qsort }
qsort [1, 2] ++ [3] ++ qsort [5, 4]
= { aplicando qsort }
(qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])
= { aplicando qsort, propriedade anterior }
([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
= { aplicando ++ }
[1, 2] ++ [3] ++ [4, 5]
= { aplicando ++ }
[1, 2, 3, 4, 5]
```

Experimentando Haskell (cont.)

- O tipo de `qsort` é

```
qsort :: Ord a => [a] -> [a]
```

- Dada uma lista `xs`, `qsort xs` é a lista formada pelos elementos de `xs` em ordem crescente.
- `qsort` implementa o algoritmo de ordenação **quick sort**.

Exercícios

Exercício 1

Dê outro cálculo possível para o resultado de `dobro (dobro 2)`.

Exercício 2

Mostre que `sum [x] = x` para qualquer número `x`.

Exercício 3

Defina uma função `product` que produza o produto de uma lista de números, e mostre, usando sua definição, que `product [2,3,4] = 24`

Exercícios (cont.)

Exercício 4

Mostre como a definição da função `qsort` deve ser modificada para que ela produza uma versão de uma lista ordenada em ordem decrescente.

Exercício 5

Qual é o efeito de trocar `<=` por `<` na definição de `qsort`?
Dica:
considere o exemplo `qsort [2,2,3,1,1]`.

Fim