UFSC / CTC / INE

Paradigmas de Programação

(Programação funcional)

Curso de Ciências da Computação: INE5416-538

Prof. Dr. João Dovicchi[1]

Florianópolis - 2008

[1]http://www.inf.ufsc.br/~dovicchi --- dovicchi@inf.ufsc.br

Conteúdo

Capítulo 1

Introdução

As linguagens funcionais são linguagens de programação com base em avaliação de funções matemáticas (expressões), evitando-se o conceito de mudança de estado com alteração de dados. Neste aspecto, este paradigma é oposto ao paradigma imperativo que se baseia em alterações de estados. Certamente, o paradigma imperativo é o mais adaptado à arquitetura de von Neumann e, apesar da programação funcional gerar um código procedimental como o modelo imperativo, o modelo declarativo da programação funcional oferece uma linguagem de alto nível voltada para um conceito bastante diferente.

As linguagens funcionais parecem estar sendo transformadas na próxima onda[6]. Palestras e tutoriais começam a aparecer mais frequentemente em conferências e convenções; livros sobre programação funcional começam a aparecer nas prateleiras; e blogs e listas de discussão são cada vez mais numerosos.

Entretanto, a divulgação linguagens funcionais têm sido confinadas no ambiente acadêmico e seu uso como linguagem de programação para o desenvolvimento de aplicativos tem sido muito aquém das possibilidades existentes.

Apesar disso, a programação funcional tem sido amplamente utilizada em diversas áreas industriais. Intel, HP, Ericsson, Motorola e Thompson são alguns exemplos de grandes empresas que a utilizam para diversas finalidades. Normalmente, as linguagens funcionais são utilizadas para testes de microprocessadores, verificações e validações de processos, geração de testes e controle de sistemas, estando mais ligadas à área de inteligência artificial.

Hoje, existem muitas linguagens funcionais . Umas puramente funcionais, tais como Haskell , Clean e Erlang, outras não tão puras permitem o uso de outros paradigmas de programação, tais como Lisp, Scheme e OCAML.

Em nosso curso de programação funcional, utilizaremos, como linguagem funcional, o Haskell e o compilador Glasgow Haskell Compiler (GHC) por

sua versatilidade e compatibilidade com bibliotecas de outras linguagens. Os interpretadores Hugs e GHCi serão usados em vários exemplos. De maneira geral, os conceitos aqui apresentados podem ser extendidos para outras linguagens funcionais como o CLEAN, CAML, ERLANG[1] etc..

A escolha do HASKELL se deve aos seguintes fatos:

- é uma linguagem de alto nível (tipagem e inferência automática);
- é uma linguagem expressiva e concisa (mais resultados com menos esforço);
- é intuitiva (construções muito próximas das declarações formais); e
- facilita a manipulação de dados complexos, combinando componentes com facilidade.

Muito embora, o HASKELL não seja uma linguagem de alto desempenho ela priorisa mais a redução do tempo dedicado à programação sobre tempo de processamento. Isto porque o desempenho está relacionado com a velocidade dos processadores e o aprimoramento das arquiteturas, que deixou de ser problema nos dias de hoje.

1.1 Histórico

A programação funcional tem sua origem no Cálculo-λ , que foi a base teórica para o desenvolvimento deste paradigma de programação. Embora o Cálculo-λ seja uma abstração matemática, sua característica de redução e substituição de argumentos e expressões formaram a base ideal para uma linguagem de programação mais formal com base na avaliação de funções. A partir do cáculo de predicados, Moses Schönfinkel e Haskell B. Curry criaram a lógica combinatória para abordar os fundamentos matemáticos e Alonso Church, baseado nos trabalhos de Schönfinkel, criou o Cálculo-λ para provar a computabilidade[2].

Embora não puramente funcional, pode-se dizer que a linguagem LISP, desenvolvida na década de 50 nos laboratórios do MIT por John McCarthy [10],

[1] Para maiores informações sobre o Clean e OCAML visite os sites: `http://www.cs.kun.nl/~clean`, `http://caml.inria.fr/` e `http://www.erlang.org/`.

[2] Alonso Church e Alan Turing chegaram à mesma conclusão por meios diferentes. Church usou o Cálculo-λ e Turing usou o conceito de máquinas abstratas. A prova da computabilidade é conhecida como tese de Church-Turing.

foi a primeira linguagem funcional, embora ela tenha sido desenvolvida como uma linguagem multi-paradigma. Mais tarde, a linguagem Scheme, simplificou e aperfeiçoou a linguagem Lisp.

Algumas fontes citam a *Information Programming Language* (IPL), desenvolvida por Allen Newell, Cliff Shaw, e Herbert Simon na década de 50, como a primeira linguagem funcional. Ela foi desenvolvida para lidar com problemas de inteligência artificial. No entanto, o caracter procedural da implementação — com um código semelhante ao Assembly — a diferencia do paradigma declarativo das linguagens funcionais de hoje.

Em 1957, Kenneth E. Iverson da Universidade de Harvard criou a APL (*A Programming Language*) com base em notação matemática e avaliação de funções com um paradigma declarativo e funcional. A APL foi implementada em 1962 e passou a ser considerada a base para a evolução das linguagens funcionais , influenciando John Backus na criação da linguagem de programação FP (*Functional Programming*), em 1977.

Na década de 70, Robert Milner da Universidade de Edimburg (Escócia) criou a linguagem funcional ML (**M***eta***L***anguage*). A linguagem ML é baseada no cálculo de predicados e Cálculo-λ polimórfico. Embora não seja puramente funcional, é considerada a mãe das linguagens puramente funcionais de hoje.

Hoje em dia, existem muitas linguagens funcionais derivadas da família ML, tais como Standard ML e Caml, incluindo F# da Microsoft. As idéias da linguagem ML influenciaram as linguagens Haskell , Clean, Cyclone e Nemerle.

A linguagem Haskell é uma linguagem puramente funcional padronizada com semântica não estrita e *open source*, definida na *Conference on Functional Programming Languages and Computer Architecture (FPCA '87)* em Portland, Oregon. Como a linguagem Miranda (1985) passou a ser a linguagem funcional mais difundida, porém proprietária, um comitê foi criado para definir uma linguagem funcional de padrões abertos. Em 1990 foi definida a linguagem Haskell na versão 1.0, mas que foi considerada estável apenas em 1997 (Haskell'98).

1.2 Linguagem Funcional - porque usar?

Como já foi dito, a programação funcional é um estilo de programação baseado em avaliação de expressões. Daí deriva o nome "funcional", uma vez que a programação é orientada para a avaliação de funções (no senso matemático e estrito da palavra). Nas linguagens funcionais, as expressões são formadas por

funções que combinam valores por relação de substituição (conforme veremos no capítulo 2).

Diferentemente das linguagens imperativas (baseadas na arquitetura de von Neumann), as linguagens funcionais podem ser classificadas como declarativas, juntamente com as linguagens lógicas.

É fácil perceber isto em um exemplo prático. Vamos tomar, por exemplo, o cálculo da soma de uma progressão aritmética de 1 a 10 com razão 1. Em uma linguagem procedural como C poderíamos escrever:

Listagem 1.1: Programa em C para cálculo da soma dos termos de uma PA de 0 a 10 e razão 1.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int soma, i;
  soma = 0;
  for (i=1; i<=10; i++){
    soma += i;
  }
  printf("%d\n", soma);

  return 0;
}
```

Em HASKELL bastaria escrever:

```
main = print (somaPA)
somaPA = sum [1..10]
```

Neste contexto, [1..10] representa uma lista de 1 a 10 e `sum` é uma função que calcula a soma dos valores de uma lista. É evidente que a função `sum` já se encontra implementada nas bibliotecas do HASKELL , mas podemos implementar uma função semelhante — a título de exemplo — construindo-a a partir da sua definição. Por exemplo, para implementar uma função `soma`, temos que definir seu tipo e implementar sua declaração:

```
soma :: forall a. (Num a) => [a] -> a
soma (x:xs) = x + soma (xs)
```

Neste exemplo, a primeira linha quer dizer:

> "Soma é definida para todo 'a' que seja numérico, recebendo uma lista de números e retornando um número."

Esta é a forma de tipagem da função e, neste caso, `a` vem da palavra inglesa *any* que significa qualquer[3]. A definição da função, propriamente dita, se encontra na linha seguinte:

> "Soma de uma lista de 'xis' e 'xises' é igual a 'xis' mais a soma de 'xises'. "

Desta forma, podemos implementar um programa semelhante, usando nossa própria definição de soma dos termos da PA.

Listagem 1.2: Programa em HASKELL para cálculo da soma dos termos de uma PA de 1 a 10 e razão 1.

```
main print(somaPA)

somaPA = soma [1..10]

soma :: forall a. (Num a) => [a] -> a
soma(x:xs) = x + soma(xs)
```

O programa da listagem 1.2 exemplifica como ficaria o código completo. Pode-se perceber, rapidamente, as vantagens da linguagem de programação funcional, não apenas no tamanho do código, mas em sua legibilidade e elegância matemática.

Além disso, outras vantagens da programação funcional estão na modularidade que este tipo de linguagem possibilita. Esta modularidade, implícita na orientação ao objeto, é nativa na linguagem funcional devido à sua capacidade de trabalhar com funções hierárquicas (*higher-order functions*) de tipagem estrita que permite o uso de classes e abrange o mecanismo de herança. As linguagens funcionais usam, ainda, um mecanismo chamado de *lazy evaluation*[4] que permite a definição de conjuntos infinitos como conjunto imagem das funções.

[3] Veremos mais sobre isto quando estudarmos "morfismo".

[4] Para traduzir *lazy evaluation* seria mais adequado chamá-la de avaliação "esperta" e não de "preguiçosa" como se poderia esperar.

Além destas vantagens, os programas escritos em linguagem funcional apresentam outras. Por exemplo, uma chamada de função não tem outra atribuição além de computar o resultado. Isto evita os efeitos colaterais e grande parte das fontes de *bugs* e, uma vez que os valores de uma expressão não podem ser alterados, ela pode ser calculada em qualquer parte do programa.

Normalmente, na programação procedural, sua natureza "imperativa" requer um fluxo definido e seqüencial de instruções. Isto não existe no estilo funcional de programação. Para os que estão acostumados com linguagens como C / C++, Pascal, Delphi, Java etc. isto pode parecer um tanto absurdo. Não é raro pensarmos em um fluxo de instruções mesmo quando usamos a linguagem funcional. Claro que, essas vantagens podem ser associadas às vantagens de outras linguagens e saber dosar as vantagens de uma ou outra linguagem no desenvolvimento de um aplicativo é essencial para o sucesso de um projeto de programa.

1.3 Funções

Com a finalidade de entender um pouco mais sobre linguagens funcionais, vamos compreender melhor o conceito de função. Genericamente, uma função é uma maneira de se relacionar dois objetos por meio de uma regra de associação. O termo função foi usado, pela primeira vez, por Leibniz em 1673.

Em álgebra, uma função corresponde a expressões de operações algébricas, enquanto que em análise numérica uma expressão define o domínio e o contradomínio de uma função. Em Cálculo-λ , uma função é um conceito primitivo no lugar ser definido em termos de teoria dos conjuntos.

Na prática, podemos dizer que uma função é um tipo de máquina que recebe uma entrada, opera sobre ela e produz uma saída. Por exemplo:

$$3 \quad \rightarrow \quad \boxed{\text{... ??? ...}} \quad \rightarrow \quad 28$$

Neste exemplo, o que sabemos é que existe uma operação sobre o número 3 que o transforma no número 28. Esta função pode ser expressa de várias maneiras. Assim, $3 + 25$ pode representar a função tão bem como $(3 \times 9) + 1$. Para podermos identificar uma forma genérica desta nossa máquina temos que saber o que acontece quando entramos outro valor, por exemplo: se entrarmos o valor 2 e nossa máquina retornar 27, podemos dizer que a função $x+25$ satisfaz as qualidades de nossa máquina. Se, no entanto, a saída for 13, podemos dizer que nossa função é $3x^2 + 1$. Vamos supor que, ao entrarmos o valor 2, obtemos

o valor 13 como resultado. Logo, podemos dizer que nossa função assume a forma genérica:

$$x \quad \rightarrow \quad \boxed{3x^2 + 1} \quad \rightarrow \quad f(x)$$

o que, matematicamente pode ser expressa na forma $f(x) = 3x^2 + 1$.

Formalmente podemos dizer que uma função de A em B é um objeto f tal que para cada $a \in A$ existe um objeto $f(a) \in B$. O conjunto A em que a função é definida é chamado de "domínio da função", enquanto que o conjunto B de valores produzidos pela função é chamado de "contradomínio da função" ou "conjunto imagem".

Naturalmente podemos dizer que uma função mapeia um determinado valor do domínio (argumento) em um valor do contra-domínio (resultado). Ou seja, mapeia os membros de um conjunto em membros de outro conjunto.

A maioria das funções mapeiam os valores um-a-um dos argumentos, entretanto, algumas funções podem mapear vários valores do domínio para um valor da imagem. Por exemplo, a função seno, por ser periódica, retorna o mesmo valor para ângulos múltiplos de 2π[5],

$$\sin(x) = \sin(2\pi + x) = \sin(4\pi + x) = ...$$

enquanto que, a função $f(x) = x^2$ retorna um valor para dois argumentos:

$$f(2) = 4 \qquad \text{e} \qquad f(-2) = 4$$

A notação de funções pode ser feita de diversas maneiras. A forma mais comum é representá-la como $f()$ mas, a mais rigorosa, do ponto de vista matemático é $f : x \rightarrow f(x)$ que especifica que f é uma função sobre um argumento x do domínio e retorna um valor $f(x)$ no contradomínio. Pode-se ainda especificar uma função de forma mais precisa como, por exemplo, $f : \Re \rightarrow \Re^+ | f(x) = x^2$. Desta forma se especifica o conjunto do domínio e da imagem da função.

Uma outra forma de especificar uma função é considerando o mapeamento do argumento no valor de retorno, por exemplo $f : x \mapsto f(x)$ uma função de x que mapeia $f(x)$.

Por exemplo, $f(x) = x^2 + x$ é equivalente a $f : (x^2 + x) \rightarrow f(x)$ e pode ser escrita como $f : \Re \rightarrow \Re^+ | f(x) = x^2$ ou, ainda, $f : (x^2 + x) \mapsto f(x)$.

[5] A rigor, a função seno retorna os mesmos valores para $\{x, \pi - x, 2\pi + x, 3\pi - x, 4\pi + x, \ldots\}$.

1.3.1 Variáveis e argumentos

O conceito de variáveis de uma função está ligado ao domínio da função. Por exemplo, uma função retornará um valor de saída, dependendo do valor de entrada, se este tiver representação no conjunto imagem. Assim, se uma função

$$f(x) = 2x^2 + 3$$

receber um valor 5 de entrada, ela sempre retornará 53 como saída. No entanto, uma função

$$f : \Re \rightarrow \Re \,|\, f(x) = \frac{\sqrt{x}}{2}$$

receber um valor -16 ela não poderá retornar um valor do conjunto imagem, uma vez que não existe um elemento dos números reais que represente o argumento do conjunto domínio. Neste caso, se o contradomínio da função incluir o conjunto dos números complexos, a função pode retornar um valor, mas para isso, o conjunto imagem deve ser definido como:

$$f : \Re \rightarrow \Im \,|\, f(x) = \frac{\sqrt{x}}{2}$$

Assim, x nos exemplos acima é chamada de variável independente e o valor que a função retorna é chamado de variável dependente. O valor que uma variável dependente recebe é chamado de argumento.

1.3.2 Funções de funções

Vamos considerar, agora, uma visão um pouco mais abstrata de função. Tomemos, por exemplo o domínio dos números naturais ($\mathbb{N}$). Tal domínio pode sofrer qualquer tipo de operação, ou seja, se quisermos representar todos os números naturais a partir de zero (ou mesmo de qualquer origem), podemos criar uma função que some a unidade a qualquer de seus elementos. Tal função pode ser representada por:

$$f(x) = x + 1 \,|x \in \mathbb{N} \tag{1.1}$$

Isto significa que, na equação 1.1 se substituirmos x por qualquer valor, a função $f(x)$ retornará o valor seguinte do conjunto, ou seja, $f(0) = 0 + 1 = 1$, $f(1) = 1 + 1 = 2$, $f(2) = 3$, ... e assim por diante.

Um outro conceito muito importante para programação funcional é o de funções de funções (funções de alta ordem). Por exemplo, se criarmos uma

função $g(x) = f(x)$ (sendo $f(x) = x+1$, conforme a equação 1.1), ao aplicarmos g em f, teremos que:

$$g(f(x)) = f(x) + 1 = (x + 1) + 1 \,|x \in \mathbb{N} \tag{1.2}$$

E se aplicarmos f em f ($f(f(x))$) e mais, se aplicarmos $f(f(f(x))) \ldots f(f \ldots (f(x)))$? Certamente, voce já percebeu que podemos criar todo o conjunto infinito de números naturais a partir do conjunto vazio: um "big bang" matemático.

O uso de funções de funções (funções hierárquicas) permite a aplicação de uma abstração matemática em outra abstração. Por exemplo, seja $f(x) = \sin(x)$ e $g(x) = \cos(x)$, podemos definir:

$$f(x, y) = \sin(x + y) = f(x)g(y) + g(x)f(y)$$

1.4 Funções recursivas

Algumas funções podem ser definidas recursivamente, por exemplo, a soma dos termos de uma PA de 0 a x é definida como:

$$f(x) = \sum_{x=0}^{x} x$$

ou, ainda, de forma recursiva:

$$f(x) = x + f(x - 1)$$

1.5 Exercícios

Exercício 1.1: Escreva as funções abaixo de forma a especificar o domínio e o contradomínio.

a. $f(x) = \cos(x)$

b. $f(x) = \log(x)$

c. $f(x) = \sqrt{x}$

Exercício 1.2: Encontre o conjunto imagem das seguintes funções:

a. $f(x) = |x| + 1$

b. $f(x) = (x+1)^2$

c.
$$h(x) = x + \frac{1}{x},\ x \in (0, +\infty)$$

d. $f(x) = -x^3 - 2x + 4$

Exercício 1.3: Encontre todos os elementos do conjunto imagem da função $f(x) = x^{\infty}$, $x \in (0, +\infty)$.

Exercício 1.4: Prove que $g(x)$ é a função inversa de $f(x)$, nos seguintes casos:

a.
$$f(x) = 2x - 1 \,\text{e}\, g(x) = \frac{x+1}{2}$$

b. $f(x) = x^{1/3}$ e $g(x) = x^3$

c.
$$f(x) = \frac{x}{x+1},\ x \in [0, +\infty) \,\text{e}\, g(x) = \frac{x}{1-x}\, x \in [0, 1)$$

Exercício 1.5: Dados:

a. $f(x) = x/2$ e $g(x) = x^2 + 4$, encontre $g(f(x))$.

b. $f(x) = 2x$ e $g(x) = x/2$, prove que $g(f(x)) = f(g(x))$.

c. $f(x) = x^2 + 1$ e $g(x) = 2x$, prove que $g(f(x)) \neq f(g(x))$.

d. $f(x) = x$, $g(x) = x^2$ e $h(x) = \sqrt{x}$, encontre $h(g(f(x)))$.

Exercício 1.6: Dado que:

$$f(x) = \frac{(1+x)}{(1-x)}$$

encontre

$$\frac{f(x)f(x^2)}{1 + (f(x))^2}$$

Capítulo 2

O Cálculo-λ

Um dos mais importantes conceitos matemáticos para a programação funcional é, com certeza, o Cálculo-λ , desenvolvido por Church e Kleene [4]. A teoria desenvolvida por eles tinha como princípio demonstrar a computabilidade efetiva de expressões.

Para a ciência da computação, o Cálculo-λ é um sistema formal da lógica matemática, desenvolvido para analisar a definição de funções, suas aplicações e a recursividade. Usando-se este tipo de formalismo matemático, qualquer função matemática computável pode ser expressa e calculada. Assim, pode-se dizer que o Cálculo-λ é uma linguagem de programação universal mínima já que é baseada numa simples regra de transformação (denominada redução) e um simples esquema de definição de função (denominada aplicação).

O mais impressionante produto das deduções de Church foi o fato de que, reduzindo-se um processo a um conjunto de regras matemáticas simples, é possível reduzir-se as chances de erro. Assim, o Cálculo-λ é considerado a base da programação correta.

O Cálculo-λ influenciou muitas linguagens de programação funcional. Lisp foi a primeira e o *pure* Lisp, que é verdadeiramente funcional, usa os conceitos de redução e aplicação do Cálculo-λ . Outras linguagens atuais — Haskell , Clean , Miranda, ML, Scheme e Erlang — foram desenvolvidas a partir do Cálculo-λ , baseadas na definição formal e na semântica denotativa que ele permite.

Em 1940, Haskell B. Curry [5] formulou a teoria da lógica combinatória como uma teoria de funções livres de variáveis. Apesar de fundamentada na teoria de funções de Schönfinkel [14] e no Cálculo-λ de Church, sua teoria foi fundamental para o desenvolvimento das linguagens funcionais.

A sintaxe do Cálculo-λ é bastante simples. Ela está fundamentada em

identificadores de variáveis, constantes, abstrações de funções, aplicação de funções e subexpressões.

2.1 Identificadores delimitados

Para compreender a teoria do Cálculo-λ , é necessário que se observe com maiores detalhes as definições das fórmulas matemáticas. Para isso vamos começar com o conceito de identificador delimitado.

Como já vimos no capítulo 1, em uma equação

$$f(x) = x^2 \tag{2.1}$$

pode-se dizer que x é uma variável e a expressão $f(x)$ assumirá valores que dependem do valor de x. Ora, esta relação é ambígua, pois se quisermos determinar se $f(x) \leq 25$ a variável x somente poderá assumir valores de 0 a 5, pois para $x > 5$, o resultado será falso. Então temos que explicitar que $x^2 \leq 25 \iff \exists x|[-5,5]\, x \in \Re$, ou seja $x^2 \leq 25$ se existir x tal que x é maior ou igual a -5 e menor ou igual a 5. Entretanto, se quisermos determinar se $f(x) < x^3$, o resultado será sempre verdadeiro, independentemente do valor que x venha a ter.

Vamos, então, observar estas entidades matemáticas com maior detalhe. Na equação,

$$\sum_{x=1}^{n} x^2 \tag{2.2}$$

x assumirá valores de $[1 \cdots n]$ e seus quadrados serão somados, ou seja, $[1 + 4 + 9 + \cdots + n^2]$. Neste caso, x pode ser chamado de uma variável dependente ou um identificador que está delimitado (de 1 a n). Note que um identificador delimitado (*bound identifier*) faz com que qualquer das expressões das equações 2.3, sejam equivalentes à equação 2.2.

$$\sum_{y=1}^{n} y^2,\ \sum_{k=1}^{n} k^2,\ \sum_{a=1}^{n} a^2, \cdots \tag{2.3}$$

Da mesma forma, na teoria dos conjuntos, podemos escrever $x|x \geq 0 = y|y \geq 0$ ou, ainda, $\forall x[x + 1 > x]$ é equivalente a $\forall y[y + 1 > y]$.

Quando temos uma expressão do tipo:

$$f(x) = x^2 + 4 \tag{2.4}$$

x	`variável`
$\lambda.M$	`abstração onde M é um termo`
(M, N)	`aplicação onde M e N são termos`

Figura 2.1: Formas dos termos no Cálculo-λ .

dizemos que o parâmetro formal x é o identificador delimitado, ou seja, a função retornará algum valor para para qualquer que seja o valor de x de forma indefinida. Pode-se perceber facilmente que para o domínio $-\infty \leq x \leq \infty$ a função retornará valores entre 4 e ∞ para o contradomínio.

2.2 Termos do Cálculo-λ

Podemos definir os termos do Cálculo-λ (λ-termo[1].) como termos construídos a partir de recursividades sobre variáveis $x, y, z, \ldots$ que podem assumir as formas descritas na figura 2.1

Em $(\lambda x.M)$, x é denominado variável dependente (*bounded variable*) e M é o termo que sofre a abstração. Cada ocorrência de x em M está vinculada à abstração. Assim, podemos escrever que:

$(\lambda x.N)M$	$\equiv$	`let x = M in N`
$(\lambda f.N)(\lambda x_1 \ldots x_k.M)$	$\equiv$	`let f x1 ... xk = M in N`

Assim, podemos representar a equação 2.4, em termos do Cálculo-λ , como:

$$\lambda x(x^2 + 4) \tag{2.5}$$

e, se $f \equiv \lambda x(x^2 + 4)$, podemos representar $f(2)$ como:

$$\lambda x(x^2 + 4)(2) \tag{2.6}$$

o que significa que cada ocorrência livre de x em $(x^2 + 4)$ será substituída por 2.

[1]Em programação usa-se `\x` para representar λx

2.3 Cópia e reescrita

O cálculo é feito substituindo-se todas as ocorrências de x por 2, ou seja, $(2^2 + 4) \Rightarrow 8$. Este tipo de operação é denominado "regra de cópia" (*copy rule*), uma vez que para $f(2)$ copiamos o valor 2 em todas as ocorrências de x. No Cálculo-λ existem apenas três símbolos: $\lambda, (,)$ e identificadores. Estes são regidos por quatro regras básicas:

1. Se x é um identificador, então x é uma expressão do Cálculo-λ ;
2. Se x é um identificador e E é uma expressão do Cálculo-λ , então λxE é uma expressão do Cálculo-λ , chamada "abstração". Assim, x é o identificador que delimita a abstração e E o corpo da abstração.
3. Se F e E são expressões do Cálculo-λ , então FE é uma expressão do Cálculo-λ , chamada "aplicação". Assim, F é o operador da aplicação e E o seu operando.
4. Uma expressão λ é produzida, sempre, pela aplicação múltipla e finita das regras 1–3.

Como estas regras operam por recursividade, podemos dizer que, uma vez que esta abstração é definida em um determinado domínio, ela pode ser aplicada em qualquer entidade. Por exemplo, digamos que queiramos aplicá-la ao domínio de z^2. Então, podemos escrevê-la como:

$$\lambda x(2x + 3)(z^2) \tag{2.7}$$

A equação 2.7 pode ser, então, re-escrita por $2z^2 + 3$. Note, por exemplo que se quisermos expressar a função de adição $f(a, b) = a + b$, podemos escrever: $\lambda ab(a + b)$ que, por sua vez, pode ser re-escrita por $\lambda a(\lambda b(a + b))$, que é denominado "currying"[2].

A esta altura voce deve estar pensando, mas que vantagens há nisto? Não seria melhor usar a velha e tradicional notação? Vejamos! Uma vez que podemos representar $f(x)$ simplesmente por fx e, sendo a aplicação da abstração sempre associativa à esquerda, podemos evitar as confusões de parênteses no caso de $f(x)(y)$, $(f(x))y$ e representar apenas por fxy. Outra vantagem é que podemos escrever a identidade como $\lambda a(a)$ e uma identidade que aplica uma combinação em si mesma como $\lambda a(aa)$. Veja, por exemplo, como uma

[2]Essa transformação é possível porque $\lambda ab(a) = \lambda a(\lambda b(a))$. O nome "currying" é usado por ter, essa expansão, sido definida por Haskell B. Curry.

combinação que inverte a ordem de dois combinadores pode, facilmente, ser expressa por $\lambda ab(ba)$.

2.4 Redução

As expressões do Cálculo-λ são computadas por operações denominadas "redução". O fundamental no Cálculo-λ é a possibilidade de se obter expressões logicamente equivalentes por meio do que se chama de redução-λ. Na verdade, a redução-λ é a combinação de 3 operações distintas de redução (β, α e η) onde a principal redução é chamada de redução-β.

Por exemplo, digamos que seja preciso avaliar a expressão:

$$\lambda y(y^2)(\lambda x(x^3+2)(2)) \tag{2.8}$$

A equação 2.8 pode ser calculada por substituições e reduções sucessivas, da seguinte forma:

$$\begin{aligned} \lambda y(y^2)(\lambda x(x^3+2)(2)) &\Rightarrow \lambda y(y^2)(2^3+2) \\ \lambda y(y^2)(2^3+2) &\Rightarrow \lambda y(y^2)(10) \\ \lambda y(y^2)(10) &\Rightarrow (10^2) \\ (10^2) &\Rightarrow 100 \end{aligned}$$

A semântica do Cálculo-λ estabelece duas regras para a redução, na computação de expressões:

1. Regra de renomeação: Uma expressão pode ser reduzida a outra pela renomeação de um identificador limitado, por qualquer outro identificador que não esteja contido na expressão.

2. Regra de substituição: Uma subexpressão da forma $(\lambda xE)A$ pode ser reduzida pela substituição de uma cópia de E na qual toda ocorrência livre de x é substituída por A, desde que isto não resulte em que qualquer identificador livre de A torne-se um delimitador.

Estas regras são conhecidas por redução-α e redução-β, respectivamente. Para compreendê-las melhor, vamos detalhá-las um pouco mais.

2.4.1 Redução-α

A redução-α (conversão α) é a substituição de nomes da variável dependente. Assim, por exemplo, a troca de $\lambda x.x$ por $\lambda y.y$ é uma conversão α, ou seja os termos são equivalentes. Entretanto, a α-redução só pode ser aplicada se não causar choque de variáveis. Por isso existem regras para que se aplique uma conversão α:

1. A troca de nome só pode ocorrer entre variáveis da mesma abstração. Por exemplo, $\lambda x.\lambda x.x$ pode ser α-convertida em $\lambda y.\lambda x.x$, mas nunca em $\lambda y.\lambda x.y$, pois esta última tem um significado totalmente diferente da expressão original.

2. A conversão α não é possível se resultar em choque de variável em uma abstração diferente. Por exemplo, se trocarmos x por y em $\lambda x.\lambda y.x$, obtem-se $\lambda y.\lambda y.y$, que não é a mesma coisa.

A conversão α pode ser exemplificada da seguinte maneira: seja uma expressão λxM que pode ser renomeada pela expressão λyN, onde N foi obtida pela renomeação do identificador x em M pelo identificador y. Isto só é possível se y não ocorrer em M. Por exemplo:

$$\lambda x(2x + z) \Rightarrow \lambda y(2y + z) \tag{2.9}$$

Uma vez que y não ocorre à direita da equação 2.9, ele pode ser usado para renomear x. O mesmo não seria possível no caso de renomear x por z, o que resultaria em erro[3].

Na prática, imagine uma função:

$$f = \lambda x(\lambda z(z + x)(4)) \Rightarrow f = \lambda x(4 + x)$$

Imagine, agora, a aplicação errada da redução-α renomeando x para z:

$$f = \lambda x(\lambda z(z + x)(4)) \Rightarrow f' = \lambda z(\lambda z(z + z)(4)) = \lambda z(8)$$

Observe que f' não é a mesma função f, pois $f(1) = 5$ e $f'(1) = 8$.

[3]Neste caso a redução resultaria em $\lambda z(2z)$.

2.4.2 Redução-β

A redução-β é a idéia de aplicação da função, onde a aplicação é usada para substituir todas as ocorrências livres da variável da aplicação.

Por exemplo, no caso da regra de redução-β vamos considerar a expressão $\lambda x(2x+1)(3)$. Neste caso, a regra de substituição tem um operador que é uma abstração e pode ser reduzido pela substituição de 3 em toda a ocorrência de x em $2x+1$, o que resulta em $2 \times 3 + 1 = 7$. No caso de

$$\lambda x(\lambda y(xy))n$$

a aplicação é uma abstração $\lambda x(\lambda y(xy))$ e usando-se a redução-β em todas as ocorrências livres de x teremos:

$$\lambda y(ny)$$

Vejamos, então a restrição da regra da redução-β. Considere a aplicação:

$$\lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) \tag{2.10}$$

Neste caso, podemos aplicar a redução:

$$\begin{aligned} \lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda z(z+y)(3))(1) \\ \lambda y(\lambda z(z+y)(3))(1) &\Rightarrow \lambda z(z+1)(3) \\ \lambda z(z+1)(3) &\Rightarrow 3+1 \\ 3+1 &\Rightarrow 4 \end{aligned}$$

Note que no primeiro passo da redução-β da equação 2.10 o identificador x foi substituído por y, então y foi substituído por 1 e, finalmente, z por 3.

Vamos, entretanto, observar o que acontece se no primeiro passo substituímos z por y por uma conversão α:

$$\begin{aligned} \lambda y(\lambda x(\lambda z(z+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda x(\lambda y(y+x)(3))(y))(1) \\ \lambda y(\lambda x(\lambda y(y+x)(3))(y))(1) &\Rightarrow \lambda y(\lambda y(y+y)(3))(1) \\ \lambda y(\lambda y(y+y)(3))(1) &\Rightarrow \lambda y(6)(1) \\ \lambda y(6)(1) &\Rightarrow 6 \end{aligned}$$

O que aconteceu? Acontece que, quando z foi substituído por y, no primeiro passo, houve uma colisão de identificadores, uma vez que y é um identificador livre em y mas não em $\lambda y(y+x)(3)$. A partir daí, houve uma mudança no significado da expressão, resultando em um valor diferente do esperado. Note a importância desta regra de redução para se evitar colisões entre os identificadores.

2.4.3 Redução-η

A conversão η usa o conceito de extensionalidade[4], ou seja, quando duas funções escritas de forma diferente resultam na mesma saída para qualquer argumento. A redução-η converte $\lambda x.Mx$ em M, se x não for livre em M.

Este tipo de conversão nem sempre é possível quando as expressões λ são interpretadas como programas.

2.5 Forma normal

Chamamos de "forma normal" a máxima redução possível de uma expressão, ou seja, a regra de substituição não pode mais ser aplicada. A computabilidade de uma expressão depende de sua forma normal ser computável, isto é, quando uma expressão atinge sua forma normal como uma resposta. A tabela 2.1 mostra alguns exemplos de expressões em sua forma não-normal e normal.

Não-normal	**Normal**
$\lambda x(x)(y)$	y
$\lambda y(fy)(a)$	fa
$\lambda x(\lambda y(xy))(f)$	$\lambda y(fy)$
$\lambda x(xx)(y)$	yy

Tabela 2.1: Exemplos de forma normal, cf. MacLennan [11].

2.6 Exercícios

1. Faça a redução das expressões:

a) $(\lambda x.x)3$

b) $(\lambda x.(xx))3$

c) $(\lambda x.(xyz))z$

d) $(\lambda x.(wy))z$

[4]Seja $f(x) = 2(x+5)$ e $g(x) = 2x+10$ dizemos que são funções extensionalmente iguais.

2. Reduza às formas normais, se possível, as seguintes expressões λ:

a) $(\lambda x.Ax)(k)$

b) $(\lambda x.Ay)(k)$

c) $(\lambda x.Ax(Kxj))(m)$

d) $(\lambda x.(\lambda x.Kxx)(j))(m)$

e) $(\lambda x.(xy))y$

f) $(\lambda x.x(xy))(\lambda u.u)$

g) $(\lambda x.xxy)(\lambda x.xxy)$

3. Prove que

$$(\lambda x.x^2)(\lambda y.(\lambda z.(z-y)a)b) \equiv a^2 + 2ab - b^2$$

4. Calcule:

$$(\lambda x.2^x)[1, 2, \cdots, 10]$$

5. Calcule as expressões λ abaixo, fazendo as reduções até a forma normal, quando possível:

a) $(\lambda x.xx)(\lambda x.xx)$

b) $(\lambda x.\lambda y.(\mathbf{add}\ y\ ((\lambda z.(\mathbf{mul}\ x\ z))8))3)5$

c) $(\lambda g.g5)(\lambda x.(\mathbf{sub}\ x\ 3))$

d) $(\lambda y.yyy)(\lambda y.yyy)$

e) $(\lambda x.\lambda y.(\mathbf{add}\ x\ ((\lambda x.(\mathbf{sub}\ x\ 7))\ y))5)6$

Nota: `add` representa adição, `mul` multiplicação e `sub` subtração.

Capítulo 3

Visão geral

A linguagem Haskell é puramente funcional fortemente tipada e *lazy*. Com relação à tipagem, significa que é estática, implícita e verificada pelo compilador, não havendo necessidade de declaração explícita. Quanto à idéia de *laziness*, podemos dizer que, em Haskell nada é feito até que seja necessário.

O sistema é formado por um compilador chamado Glasgow Haskell Compiler (GHC), um interpretador GHC interativo (GHCi) e vários módulos que contêm tipos, classes e funções pré-definidas. O módulo *default* é chamado de "Prelude" e é importado pelo compilador ou interpretador sempre que são utilizados. Existem outros interpretadores e compiladores da linguagem e, dentre eles, o Hugs é apenas interpretador e o NHC98 é apenas compilador.

O GHC é o compilador padrão da linguagem Haskell e seu ambiente interativo (GHCi) é mais lento que o Hugs para carregar arquivos, mas permite definições de funções dentro do ambiente.

O Hugs é um interpretador bastante rápido para carregar os arquivos, porém mais lento que GHCi para executá-los, implementando todas as extensões do Haskell '98 (padrão) embora não suporte todas as bibliotecas. Além disso, suporta interface gráfica; é escrito em C e funciona em todas as plataformas.

O NHC98 é apenas compilador; é menos utilizado mas produz um código menor e mais rápido que o GHC, suportando o padrão Haskell '98 e algumas extensões.

3.1 Glasgow Haskell Compiler

O GHC pode ser obtido em `http://haskell.org/ghc` e é um compilador bastante estável com muitas opções e diretivas de otimização e compilação.

O código pode ser gerado em binário para a plataforma utilizada ou em linguagem C. O GHC implementa diversas extensões da linguagem padrão, tais como concurrência, interface com outras linguagens, classes de tipos multi-parametrizados, escopo de tipos de variáveis etc.. O GHC possui coletor de lixo da memória e perfilador de tempo e espaço.

A compilação pode ser feita em linha de comando ou em ambiente de desenvolvimento com suporte ao HASKELL [1]. O ambiente Kdevelop do KDE suporta a linguagem HASKELL . O utilitário "make" dos Unices podem ser utilizados para automatizar a compilação.

Em linha de comando, a compilação é bastante fácil. Supondo que o programa tenha a função principal (`main`) em um arquivo `Main.hs`, a linha de comando para compilá-lo pode ser:

```
$ ghc --make Main.hs -o main
```

O comando `ghc` chama o compilador; a opção `--make` diz ao compilador que `Main.hs` é um programa a ser compilado e a opção `-o` (*output*) fornece o nome de saída do executável gerado.

Existem muitas opções de compilação, por exemplo:

- `-fglasgow-exts` Permite uso das *Glasgow extensions*
- `-i<dir>` inclui diretórios para importação de módulos
- `-O<n>` Permite níveis de otimização de código
- `-H32m` Amplia o tamanho do *heap* para 32M
- `-cpp` Processa código de C++

Listagem 3.1: Exemplo de inclusão de diretivas no código do programa.

```
{-# OPTIONS -fglasgow-exts #-}
module Foo (main) where

import ...

main = ...
```

[1] `http://www.haskell.org/haskellwiki/IDEs`

Existem muitas outras diretivas de compilação. Estas diretivas podem ser inseridas no código fonte dos programas em HASKELL (ver listagem 3.1).

Normalmente, o compilador emite vários tipos de alertas (*warnings*) sendo que estas opções podem ser controladas na linha de comando. Os alertas, na compilação padrão, são:

- `-fwarn-overlapping-patterns`,
- `-fwarn-deprecations`,
- `-fwarn-duplicate-exports`,
- `-fwarn-missing-fields` e
- `-fwarn-missing-methods`.

Além destas, o uso de `-W` emite, além dos alertas padrão, mais o alerta `-fwarn-incomplete-patterns` e a opção `-w` desliga todos os alertas. A opção `-Wall` emite todos os tipos de alertas.

Mais informações podem ser encontradas no *site* do GHC, na área de documentação: `http://haskell.org/haskellwiki/GHC:Documentation`.

3.2 GHCi

Para testar as funções e os conceitos, podemos utilizar um interpretador da linguagem HASKELL . O GHC já apresenta um interpretador quando invocado, na linha de comando como `ghci` ou `ghc -interactive`.

O interpretador apresenta uma linha de comandos (*shell*) onde se pode digitar comandos ou testar funções. Ao ser carregado, o interpretador e os pacotes necessários, termina com um novo sinal de prompt (`Prelude>`):

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.04, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Loading package haskell98 ... linking ... done.
Prelude>
```

Uma vez no interpretador, os comandos devem começar com ":", por exemplo, módulos e programas podem ser carregados com o comando `:load` ou, simplesmente `:l`. O interpretador compilará o módulo carregado, interpretará o módulo e carregará as funções, apresentando um novo prompt (`*Main>`). Para saber quais os tipos definidos nas funções, pode-se usar o comando `:browse`.

3.3 Usando o compilador e o interpretador

Tomemos um exemplo, com o propósito de descrever o uso do compilador e do interpretador[2]. Vamos considerar um problema onde precisamos encontrar o maior número entre três algarismos inteiros. Assim, a entrada do programa deve ser de três algarismos do tipo inteiro. Vamos, em seguida, dar um nome para a nossa função de `maxTres` que tem o tipo:

```
Int -> Int -> Int -> Int
```

Os três primeiros tipos são relativos à "entrada" da função e o último ao resultado, ou seja a função recebe três argumentos inteiros como entrada e retorna um valor inteiro.

Em nosso problema, o primeiro passo poderia ser comparar dois números inteiros para obter o maior dentre eles. Para isso, podemos criar uma função `maxi` para efetuar esta computação:

```
maxi a b | a>=b      = a
         | otherwise = b
```

O sinal "`|`" serve para separar condições e devem estar posicionados um sobre o outro. Assim, dizemos que `maxi a b` é igual a `a` caso $a \geq b$, caso contrário (*otherwise*) é igual a `b`.

A solução de nosso problema requer que `maxTri` compute `maxi a (maxi b c)` (usando o conceito de funções hierárquicas) assim, o algoritmo seria escrito:

```
maxTres a b c = maxi a (maxi b c)
```

A generalização do problema pode ser extendida para a computação do máximo de uma lista de inteiros. Assim, uma função mais genérica, que podemos chamar de `maxList` seria do tipo:

[2]No capítulo 4 descrevermos as características léxicas e sintáticas da linguagem

```
maxList :: [Int] -> Int
```

Isto quer dizer que a função `maxList` recebe como argumento uma lista de inteiros e retorna um inteiro. Uma das primeiras preocupações do programa é: o que fazer com uma lista vazia? Digamos que uma lista vazia retorne 0 e uma lista não vazia tenha outras implicações. Neste caso, teremos que considerar qualquer posição do maior elemento dentro da lista. Uma lista pode ser descrita como formada de uma "cabeça" x e uma "cauda" xs que é representada por $(x : xs)$. Assim, podemos definir nossa função como:

```
maxList [] = 0
maxList (x:xs) = maxi x (maxList xs)
```

Podemos ver, aqui, de que maneira podemos reutilizar uma função previamente definida. Assim, o programa todo poderia reaproveitar o código escrito para calcular outras funções:

Listagem 3.2: Exemplo de programa: Maxi.hs

```
module Maxi (
        maxi,
        maxiTri,
        maxiLst
        ) where

maxi :: Ord a => a -> a -> a
maxi a b | a>=b      = a
         | otherwise = b

maxiTri :: Ord a => a -> a -> a -> a
maxiTri a b c = maxi a (maxi b c)

maxiLst :: (Ord a, Num a) => [a] -> a
maxiLst [] = 0
maxiLst (x:xs) = maxi x (maxiLst xs)
```

Uma vez construido o módulo com as funções (ver listagem 3.2) podemos utilizá-las em qualquer programa que necessite destas funções, que pode ser compilado junto com este módulo. Por exemplo, um programa para determinar o maior valor dos elementos de uma lista dada poderia ser escrito como:

Listagem 3.3: Exemplo de programa para determinar o maior valor dos elementos de uma lista.

```
module Main (main) where

import Maxi (maxiLst)

main = print (maxiLst [30,29,54,34,95,12,49])
```

Observe na listagem 3.3 que o módulo `Main` importa o módulo `Maxi` e deste, importa apenas a função necessária (`maxiLst`).

Os módulos podem ser carregados em um interpretador e a construção de suas funções podem ser testadas.

Para testar as funções e os conceitos, neste curso, vamos utilizar um interpretador da linguagem HASKELL . Para testá-los os voce pode carregá-los em um interpretador como o **GHCi** ou **Hugs**. Para isso recomendamos instalar o GHC (Glasgow Haskell Compiler) que instala o interpretador[3].

Para este procedimento, digite o código em um editor de texto plano (joe, vi, pico etc.) salve o arquivo com a extensão ".hs" (por exemplo: `maxil.hs`) e inicie o interpretador na linha de comandos, digitando `ghci`:

```
-bash$ ghci
```

Verá que será carregado o interpretador e os pacotes necessários, terminando com um novo sinal de prompt (`Prelude>`):

```
            ___         ___ _
           / _ \ /\  /\/ __(_)
          / /_\// /_/ / /  | |      GHC Interactive, version 5.04, for Haskell
98.
         / /_\\/ __  / /___| |      http://www.haskell.org/ghc/
         \____/\/ /_/\____/|_|      Type :? for help.

         Loading package base ... linking ... done.
         Loading package haskell98 ... linking ... done.
         Prelude>
```

Carregue o programa usando o comando :load ou :l, por exemplo:

[3]Neste curso utilizaremos um terminal de um sistema Unix (o FreeBSD) para as atividades e práticas dos alunos.

```
Prelude> :l Maxi
Compiling Maxi              ( Maxi.hs, interpreted )
Ok, modules loaded: Maxi.
*Maxi>
```

O interpretador compilará o módulo carregado, interpretará o módulo e carregará as funções, apresentando um novo prompt (*Main¿). Caso voce deseje saber quais os tipos definidos nas funções, voce pode usar o comando :browse:

```
*Maxi> :browse *Maxi
maxiTri :: forall a. (Ord a) => a -> a -> a -> a
maxi :: forall a. (Ord a) => a -> a -> a
maxiLst :: forall a. (Ord a, Num a) => [a] -> a
*Maxi>
```

Finalmente, experimente o programa para ver se tudo está funcionando como deveria, por exemplo, digite:

```
*Main> maxiLst [3,7,4,53,9,0,1]
53
*Main>
```

Repita os testes com alguns argumentos diferentes e experimente calcular usando uma lista vazia (`maxiLst []`).

Compilar nada mais é do que juntar as idéias... Afinal, resolvendo um determinado problema por meio de um programa de computador o que se faz, via de regra, é "fracionar" a solução e, depois, juntar as partes, as funções e as rotinas ou procedimentos.

Em C, por exemplo, juntamos partes de código (.c) com bibliotecas ou cabeçalhos (.h) e "linkamos" os objetos (.o) em um programa que possa ser executado pela máquina e que produza a solução necessária para o referido problema.

Para isto, é necessário um outro programa chamado compilador. Cada linguagem tem seus compiladores e existem vários compiladores de linguagem funcional. Muitos deles já se encontram em estágios bastante avançados, enquanto outros ainda são experimentais. Neste nosso curso, como já afirmamos, vamos focar a linguagem Haskell e seu compilador mais flexível, o Glasgow Haskell Compiler (GHC)[4]. Nada impede que se use outros compiladores como o

[4]Página *Web* do GHC: `http://www.haskell.org/ghc/`

Clean que é outro compilador de linguagem funcional excelente, muito estável e poderoso. Como já justificamos antes, a escolha do Haskell se deu por contingência didática, uma vez que a sua sintaxe é bastante fácil e clara, além de ser muito próxima do Cálculo Lambda.

Compilar requer alguns truques para que o código gerado seja rápido, eficiente e enxuto. Qualquer compilador, de qualquer linguagem, oferece opções de compilação que vão desde a compilação estática (com todas as bibliotecas de funções incluídas no código binário final) até a compilação que usa bibliotecas dinâmicas (por ex.: ELF no Linux ou DLL em outro sistema). A grande flexibilidade de opções de compilação do GHC permite a ligação (linking) com as bibliotecas e/ou suas partes.

O compildor GHC pode ser invocado de várias maneiras. Quando invocado com a opção –interative corresponde ao comando "ghci" e o módulo é carregado no interpretador. Quando invocado com a opção –make o compilador compila vários módulos, checando as dependências automaticamente. Outra maneira invocá-lo com as opções de compilação ou, simplemente. com a opção de arquivo de saída e o arquivo a ser compilado:

```
ghc -o <arquivo_de_saída> <arquivo_de_entrada>
```

A compilação em linha de comando pode ser feito em etapas:

```
$ ghc -c -O2 Maxi.hs
$ ghc -c -O2 maxil.hs
$ ghc -o maxil maxil.o Maxi.o
```

Ao se invocar o compulador com a opção `--make` ele carrega todas as depdências necessárias de forma automática:

```
ghc --make -O1 maxil.hs
```

Para isso, o nome do arquivo e do módulo tem que ser exatamente igual. Neste caso, por exemplo, o nome do módulo é `Maxi` e o arquivo é `Maxi.hs`. Note que o Haskell é *case sensitive*.

3.4 Exercícios

Listagem 3.4: Modulo Fatorial (Fact.hs).

```
module Fact (fac) where

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

Listagem 3.5: Modulo Combinatória (Comb.hs).

```
module Comb (arSim, arRep,
             perSim, perCirc,
             combSim, combRep) where
import Fact

-- arranjo simples
arSim m p  = fac m / fac (m-p)
-- arranjo c/ repetição
arRep m p  = m ** p

-- permutação simples
perSim m   = fac m
-- permutação circular
perCirc m  = fac (m-1)

-- combinação simples
combSim m p = fac m / (fac (m-p) * fac p)
-- combinação c/ repetição
combRep m p = combSim (m+p-1) p
```

Listagem 3.6: Programa exemplo para Análise Combinatória (ancomb.hs).

```
module Main (main) where
import Comb
import Fact

main = print (arSim 4 2)
```

Exercício 1: Usando o módulo Comb.hs (listagem 3.4), teste-o no interpretador.

Exercício 2: Compile o programa exemplo (listagem 3.6) e teste o executável.

Exercício 3: Modifique o programa exemplo para calcular:

a. $A_s(6, 3)$ (Arranjo simples)

b. $A_r(4, 2)$ (Arranjo com repetição)

c. $P_c(8)$ (Permutação cíclica)

d. $C_r(10, 4)$ (Combinação com repetição)

Capítulo 4

Análise léxica

A análise léxica da linguagem HASKELL não é um problema trivial e, embora a linguagem tenha sido inicialmente baseda no flex (The Fast Lexical Analyser)[1], as expressões regulares foram ficando cada vez mais complexas devido às diversas particularidades da linguagem. Por exemplo, regras de *layout*, precedência e associatividade de operadores etc. que precisam de um conjunto de funções que possam classificar os elementos de cada conjunto particular precisaram ser definidas para semânticas apropriadas.

Além disso, a linguagem pode ser definida usando-se uma extensão do Cálculo-λ e um conjunto de operadores primitivos [15]. As primitivas são funções de baixo nível em HASKELL que refletem as noções básicas de aritmética e o ambiente de operações.

Cada primitiva tem que:

- Isolar seus argumentos, na forma de uma lista ligada
- Avaliar os argumentos na sua forma normal
- Verificar se os argumentos são de tipos corretos
- Criar um *node* para o tipo apropriado
- Preencher o campo do *node* com o resultado
- Retornar o *node*

Por questões de compatibilidade usaremos as seguintes convenções, conforme o *Haskell online report* (`http://www.haskell.org/onlinereport/lexemes.html`):

[1] `http://flex.sourceforge.net/`

- *[valor]* optional
- {*valor*} nenhuma ou mais repetições
- *(valor)* agrupamento
- $val_1|val_2$ escolha (ou)
- $val_{<val'>}$ elementos generados por val exceto gerados por val'.
- `sintaxe` sintaxe terminal em fonte typewriter

4.1 Estrutura léxica do programa

De acordo com o *Haskell online report*, a estrutura léxica de um programa em HASKELL pode ser descrita como:

```
program -> {lexeme | whitespace }
  lexeme -> qvarid | qconid | qvarsym | qconsym |
            literal | special | reservedop | reservedid
    literal ->integer | float | char | string
    special ->( | ) | , | ; | [ | ] | `| { | }
  whitespace -> whitestuff {whitestuff}
    whitestuff ->whitechar | comment | ncomment
      whitechar -> newline | vertab | space | tab | uniWhite
        newline -> return linefeed | return | linefeed | formfeed
          return -> a carriage return
          linefeed -> a line feed
          formfeed -> a form feed
        vertab -> a vertical tab
        space -> a space
        tab -> a horizontal tab
        uniWhite -> any Unicode character defined as whitespace
      comment -> dashes [ any<symbol> {any}] newline
        dashes -> -- {-}
        any -> graphic | space | tab
      ncomment -> opencom ANYseq {ncomment ANYseq}closecom
        opencom -> {-
        closecom -> -}
        ANYseq -> {ANY}<{ANY}( opencom | closecom ) {ANY}>
          ANY -> graphic | whitechar
```

```
graphic -> small | large | symbol | digit | special | : | " | '
  small -> ascSmall | uniSmall | _
    ascSmall -> a | b | ... | z
    uniSmall -> any Unicode lowercase letter
  large -> ascLarge | uniLarge
    ascLarge -> A | B | ... | Z
    uniLarge ->  {uppercase | titlecase} Unicode letter
  symbol -> ascSymbol | uniSymbol<special | _ | : | " | '>
    ascSymbol -> ! | # | $ | % | & | * | + | . | / | < |
                 = | > | ? | @ | \ | ^ | | | - | ~
    uniSymbol -> {Unicode symbol | punctuation}
  digit -> ascDigit | uniDigit
    ascDigit -> 0 | 1 | ... | 9
    uniDigit -> Unicode decimal digit
  special -> {octit | hexit}
    octit -> 0 | 1 | ... | 7
    hexit -> digit | A | ... | F | a | ... | f
```

4.1.1 Comentários

Em HASKELL os comentários iniciam com os caracteres `--{-}` (dois ou mais traços) e terminam no final da linha. Comentários multi-linhas começam com `{-` e terminam com `-}`. Os comentários podem ser aninhados, tais como:

```
{-
  Comentário multi-linha
   {- comentário aninhado -}
-}
```

Mas deve-se tomar o cuidado de fechar cada token aberto para comentário. comentários aninhados podem ser usados como pragmas do compilador como, por exemplo:

```
factorial :: Num a => a -> a
factorial 0 = 0
factorial n = n * factorial (n-1)
{-# SPECIALIZE factorial :: Int -> Int,
```

```
factorial :: Integer -> Integer #-}
```

Desta forma, o compilador entenderá que a chamada da função fatorial pode ser feita para parâmetros de tipo `Int` ou `Integer`, evitando-se erro com operações numéricas (*numeric overflow*).

4.1.2 Identificadores

em HASKELL um identificador é formado por uma letra que pode ser seguida de outras letras, números, aspas simples ou o caractere _. Podem ser de dois tipos:

1. Identificadores de variáveis: que iniciam, obrigatoriamente, com letra minúscula ou o caractere _. Por exemplo: `var, _VAR, func, fUNC` etc..
2. Identificadores de construtores: iniciam com letra maiúscula. Por exemplo: `Int, Integer, Bool, Char` etc..

O traço de sublinhado (*underscore*) é tratado como letra minúscula quando associado a nomes de identificadores. No entanto o sinal _ quando sozinho é tratado como um identificador especial, significando um caractere "coringa" (*wildcard*). Identificadores que iniciam com este caractere não são verificados pelo compilador, permitindo o seu uso como parâmetros não utilizados.

Alguns identificadores são reservados: `case | class | data | default | deriving | do | else | if | import | in | infix | infixl | infixr | instance | let | module | newtype | of | then | type | where | _`

Um identificador pode ser qualificado. Por exemplo, considere um programa (listagem 4.1) que importa um módulo (listagem 4.2). Neste caso o programa principal importa apenas a função `func2` do módulo `Funcs` e, no programa principal a função pode ser referenciada apenas como `func2` ou pode ser qualificada como `Funcs.func2`.

Listagem 4.1: Programa principal.

```
module Main (main) where
import Funcs (func2)

main = print (func2 ...)
```

Listagem 4.2: Módulo.

```
module Funcs (func1, func2, ...) where

func1 = ...

func2 = ...

...
```

O parser do compilador entende o ponto como um construtor de qualificação, quando o primeiro *token* inicia com maiúscula. Por exemplo: `f.g` é interpretado como 3 *tokens* (`f . g`) e `F.g` é interpretado como "g qualificado" (`F.g`).

O qualificador mantém o mesmo tratamento sintático de um elemento. Por exemplo, O operador `Prelude.+` é um operador de adição (`+`) com o tratamento sintático definido no módulo `Prelude`, isto é, a fixicidade e a precedência deste operador descritas no módulo são mantidas.

4.1.3 Operadores

Os operadores são formados por um ou mais caracteres de símbolos (`+, -, *` etc.). Operadores que iniciam com ":" são construtores e os demais são operadores comuns. Os operadores `.. | : | :: | = | \ | | | <- | -> | @ | ~ | =>` são operadores reservados.

- O operador ".." define uma PA em uma lista implícita. Por exemplo, `[1..10]` é uma PA de 1 a 10 de razão 1.
- O operador ":" em si é usado apenas como construtor de listas. Por exemplo: `1:2:3:4:[]` gera a lista `[1,2,3,4]`.
- O operador "::" é usado para definir tipos (pode ser lido como "é do tipo"). Por exemplo:

```
func   :: Int -> Int
5      :: Int
'a'    :: Char
"nome" :: [Char]
12     :: Double       -- saída 12.0
```

- O operador "`=`" corresponde à igualdade matemática.
- O operador "\" representa a letra λ em expressões do Cálculo-λ . Por exemplo: `\x->x` corresponde a $\lambda x.x$.
- O operador "`|`"tem duas interpretações: em uma expressão é um *gard*, por exemplo:

```
f x | x > 0   = 1
    | x == 0  = 0
    | x < 0   = -1
```

  em uma expressão dentro de uma lista tem o significado de "tal que". Por exemplo: `[x*x | x <- [1,2,3]]` (leia-se x vezes x, tal que x recebe a lista $\{1, 2, 3\}$. Neste caso, a expressão retorna a lista `[1,4,9]`.
- O operador "`<-`" atribui valores a um argumento. Por exemplo, `[x | x <- [1,5,12,3,23,11,7,2], x>10]` passa para os argumentos para x que casam com padrão (> 10).
- O operador "`->`" tem dois significados. Em uma expressão do Cálculo-λ representa o "ponto", por exemplo: `\x->x` é igual a $\lambda x.x$. O outro uso do operador é na atribuição de tipos: `func :: [Int] -> Int` (`func` é uma função que recebe uma lista de inteiros e retorna um inteiro.
- O operador "`@`" é usado como definição de padrão "como". Por exemplo:

```
main = print (dup1st [2,3,4])

dup1st y@(x:xs) = x:y
```

  No exemplo acima, a função gera uma lista formada do primeiro elemento de `(x:xs)` com `y` onde `y` é `(x:xs)`, retornando a lista: `[2,2,3,4]`.
- O operador "`~` " é chamado de *lazy pattern* e será discutido mais adiante.
- Finalmente, o operador "`=>`" define a classe de argumentos e retorno de tipos. Por exemplo:

```
realToFrac   :: (Real a, Fractional b) => a -> b
```

`(Real a, Fractional b)` limita a entrada da função a um argumento da class `Real` e o retorno da função a um valor da classe `Fractional`.

4.1.4 Indentação e pontuação

Os programas em HASKELL podem ser pontuados e ou indentados para definir a sintaxe. É permitida a omissão de chaves ({ }) e pontuação (;) sempre que se respeitar o *layout* de indentação. Por exemplo:

```
f x = let {x = 1; y = 2; g z = expr2} in expr1
```

é equivalente a:

```
f x = let x   = 1
          y   = 2
          g z = expr2
      in expr1
```

4.2 Expressões

Em HASKELL os principais elementos de um programa são as expressões. Uma vez que HASKELL é uma linguagem declarativa, as expressões têm um tipo estático e retornam um valor dependendo dos argumentos utilizados. A forma genérica de uma expressão em HASKELL é:

```
nome param {param} = expr
```

onde `expr` é uma expressão que contém o(s) parâmetro(s), números e operadores aritméticos.

Os operadores nas expressões em HASKELL são interfixos (*infix operators*) e seguem as regras aritméticas de associação (levo-associativos, dextro-associativos e homo-associativos). Além disso, a associatividade tem precedência de de 0 a 9 (ver tabela 4.1)

Preced.	**Levo-ass.**	**Homo-ass.**	**Dextro-ass.**
9	`!!`		`.`
8			`^, ^^, **`
7	`*, /, ‘div‘, ‘mod‘, ‘rem‘, ‘quot‘`		
6	`+, -`		
5			`:,++`
4		`==, /=, <, <=, >, >=, ‘elem‘, ‘notElem‘`	
3			`&&`
2			`||`
1	`>>,>>=`		
0			`$, $!,‘seq‘`

Tabela 4.1: Tabela de precerência e associatividade dos operadores.

A negação é o único operador prefixado (*prefix operator*) em HASKELL e tem a mesma precedência do operador "`-`" definido no módulo `Prelude`. Mesmo assim, qualquer operador pode ser prefixado se usado entre parênteses, por exemplo

```
f x = (+) x 2          -- equivale a f x = x + 2
g x = (+) ((*) x x) 2  -- equivale a g x = x*x+2
```

Ambiguidades geradas por abstrações λ, expressões `let` e condicionais são resolvidas pela meta-regra que cada um dos construtores abrange o mais à direita (dextro-associação) possível (veja os exemplos na tabela 4.2).

Expressões que podem causar conflito de interpretação, como no caso de uso de operadores não associativos (homo-associativo), devem ser explicitadas com parênteses para evitar a intepretação errada pelo *parser* do GHC.

Notação	**Interpretação**
`f x + g y`	`(f x) + (g y)`
`- f x + y`	`(- (f x)) + y`
`let ... in x + y`	`let ... in (x + y)`
`z + let ... in x + y`	`z + (let ... in (x + y))`
`f x y :: Int`	`(f x y) :: Int`
`\x -> a+b :: Int`	`\x -> ((a+b) :: Int)`

Tabela 4.2: Exemplos de *parsing* de expressões. (fonte: *The Haskell 98 Report*)

4.3 Exercícios

Exercício 1: Indente corretamente os exemplos abaixo:

a.
```
if foo
then do acao1
acao2
acao3
else do acao4
```

b.
```
do primeira_coisa
if condicao
then isto
else aquilo
segunda_coisa
```

Exercício 2: Converta os operadores para prefixados:

a. `f x = 3*x + 2`

b. `junta a b = a ++ b`

Capítulo 5

Classes e Tipos

A linguagem HASKELL tem diversos tipos de dados já definidos no módulo `Prelude` que é importado para todos os programas. Os tipos de dados são derivados de classes que permitem a aplicação de operadores e seu escopo em expressões.

5.1 Classes

A linguagem HASKELL define várias classes que delimitam as operações sobre tipos de dados que são instanciados por ela. Uma declaração de classe define uma nova classe e os métodos dela.

A declaração de classes tem a forma genérica

```
class <Nome_da_classe> a => <Nome_nova_classe> a where

declaracao1 [ ... declaracaoN]
```

Por exemplo, a classe `Real` é definida como vinculada às classes `Num` e `Ord` e declarada com uma função `toRational` que recebe um valor e retorna um número `Racional`.

```
class (Num a, Ord a) => Real a  where
    toRational  ::  a -> Rational
```

Em HASKELL , as classes definem métodos ou conjuntos de operações sobre tipos. Um tipo pode ser uma instância de uma classe e herdar o método correspondente a cada operação sobre ele. Além disso, as classes podem ser organizadas de forma hierárquica, dando a idéia de superclasses e subclasses, que por sua vez permitem a transmissão de herança de métodos. No entanto, é bom ressaltar que, não se deve confundir tipos com objetos, uma vez que a linguagem HASKELL não possui a noção de objeto ou de alteração de estado, fazendo com que os métodos em HASKELL sejam mais seguros por ser mais fortemente tipados. Qualquer tentativa de aplicar um método a um valor cujo tipo não pertence à classe, será detectado pelo compilador.

Existem várias classes definidas no módulo `Prelude` e outras em módulos específicos. As principais são as classes `Eq, Ord, Enum, Read, Show, Functor, Monada` e `Bounded` e, ainda as classes numéricas (`Num, Integral, Real, Fractional, Floating, RealFrac` e `RealFloat`). A figura 5.1 mostra a hierarquia entre as classes em HASKELL .

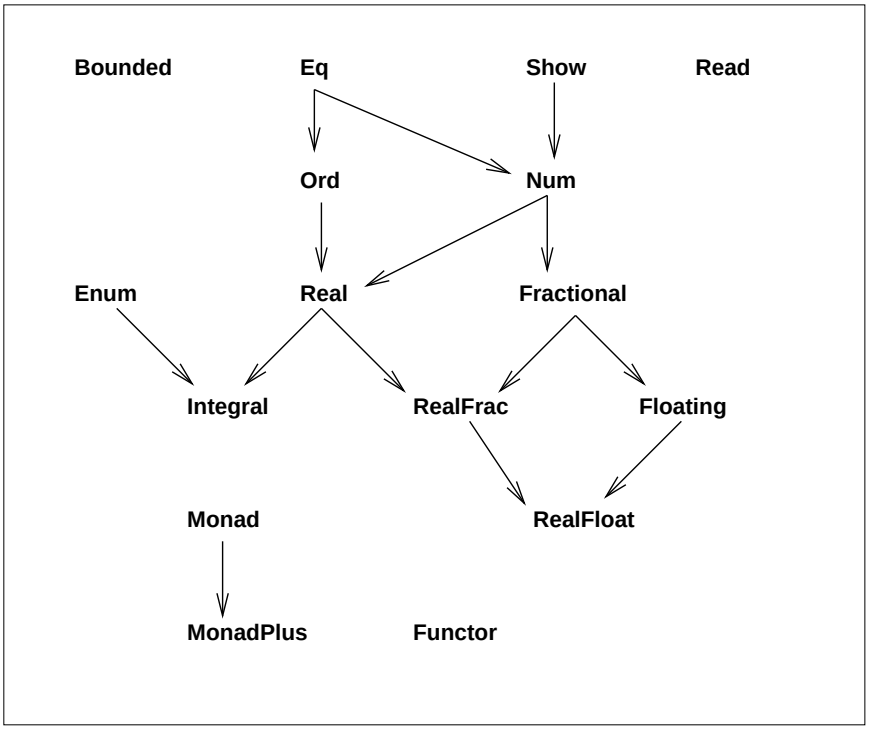


Figura 5.1: Hierarquia das classes em HASKELL .

5.1.1 Classes primitivas

As classes primitivas são classes definidas primariamente, isto é, não são definidas a partir de outras classes. São elas:

- `Bounded` é uma classe que estabelece limites mínimos e máximos e que instancia os tipos `Int, Char, Bool, (), Ordering` e `tuples`. A definição formal é:

```
class  Bounded a  where
 minBound         :: a
 maxBound         :: a
```

  onde `a` (*any*)[1] tem um delimitador mínimo e outro máximo.

- `Eq` que trata os métodos de igualdade e desigualdade e instancia todos os tipos, com excessão dos tipos `IO` e funções. É formalmente definida como:

```
class  Eq a  where
    (==), (/=)       :: a -> a -> Bool

        -- Minimal complete defintion:
        --      (==) or (/=)
    x /= y           =  not (x == y)
    x == y           =  not (x /= y)
```

- `Enum` que trata a enumerabilidade, ou seja, define os métodos de operações sobres tipos seqüencialmente ordenados instanciando os tipos `(), Bool, Char, Int, Integer, Ordering, Float` e `Double`. Sua definição formal é:

```
class  Enum a  where
  succ, pred       :: a -> a
  toEnum           :: Int -> a
  fromEnum         :: a -> Int
  enumFrom         :: a -> [a]             -- [n..]
```

[1]Em HASKELL o nome `a` é derivado do inglês *any* e denota um argumento polimórfico delimitado por uma classe.

```
  enumFromThen     :: a -> a -> [a]        -- [n,n'..]
  enumFromTo       :: a -> a -> [a]        -- [n..m]
  enumFromThenTo   :: a -> a -> a -> [a]   -- [n,n'..m]

      -- Minimal complete definition:
      --      toEnum, fromEnum
  succ           =  toEnum . (+1) . fromEnum
  pred           =  toEnum . (subtract 1) . fromEnum
  enumFrom x     =  map toEnum [fromEnum x ..]
  enumFromTo x y =  map toEnum [fromEnum x .. fromEnum y]
  enumFromThenTo x y z =
                    map toEnum [fromEnum x, fromEnum y .. fromEnum z]
```

- `Show` e `Read` são classes que definem os métodos de conversão entre números e caracteres e vice-versa. Estas classes instanciam todos os tipos, exceto `IO` e funções. A definição formal da classe `Show` é:

```
class  Show a  where
  showsPrec        :: Int -> a -> ShowS
  show       :: a -> String
  showList         :: [a] -> ShowS

-- Mimimal complete definition:
-- show or showsPrec
  showsPrec _ x s   = show x ++ s

  show x        = showsPrec 0 x ""

  showList []     = showString "[]"
  showList (x:xs) = showChar '[' . shows x . showl xs
                    where showl []     = showChar ']'
                          showl (x:xs) = showChar ',' . shows x .
                                         showl xs
```

  e a definição formal da classe `Read` é:

```
class  Read a  where
  readsPrec        :: Int -> ReadS a
  readList         :: ReadS [a]

-- Minimal complete definition:
```

```
-- readsPrec
  readList = readParen False (\r -> [pr | ("[",s)  <- lex r,
                                          pr       <- readl s])
             where readl  s = [([],t)   | ("]",t)  <- lex s] ++
                              [(x:xs,u) | (x,t)    <- reads s,
                                          (xs,u)   <- readl' t]
                   readl' s = [([],t)   | ("]",t)  <- lex s] ++
                              [(x:xs,v) | (",",t)  <- lex s,
                                          (x,u)    <- reads t,
                                          (xs,v)   <- readl' u]
```

- `Monad` é uma classe que define o método de operações sobre mônadas[2], instanciando os tipos `IO, Maybe` e `[]`. Sua definição formal é:

```
class  Monad m  where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

        -- Minimal complete definition:
        --      (>>=), return
    m >> k  =  m >>= \_ -> k
    fail s  = error s
```

- `Functor` define os métodos para tipos que podem ser mapeados, por isso instancia tipos `[], IO` e `Maybe`. É definida como:

```
class  Functor f  where
    fmap  :: (a -> b) -> f a -> f b
```

5.1.2 Classes secundárias

Aqui denomina-se classes secundárias às classes definidas a partir de outras classes. São elas:

[2]Entidades matemáticas definidas por relações homológicas e será estudada mais adiante no capítulo sobre I/O.

- `Ord` define métodos para tipos de dados totalmente ordenados, instanciando todos os tipos, exceto funções, `IO` e `IOError`. Sua definição formal é:

```
class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    -- Note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

- `Num` define os métodos para operações com números e instancia os tipos `Int, Integer, Float` e `Double`. É definida como:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)    :: a -> a -> a
    negate           :: a -> a
    abs, signum      :: a -> a
    fromInteger      :: Integer -> a

        -- Minimal complete definition:
        --      All, except negate or (-)
    x - y            =  x + negate y
    negate x         =  0 - x
```

- `Real, Integral, Fractional, Floating, RealFrac` e `RealFloat` definem os métodos numéricos de operações. `Real` instancia os tipos `Int,`

`Integer`, `Float` e `Double`, as outras classes instanciam apenas `Float` e `Double`. São definidas como:

```
class  (Num a, Ord a) => Real a  where
    toRational ::  a -> Rational

class  (Real a, Enum a) => Integral a  where
    quot, rem, div, mod :: a -> a -> a
    quotRem, divMod     :: a -> a -> (a,a)
    toInteger           :: a -> Integer

class  (Num a) => Fractional a  where
    (/)          :: a -> a -> a
    recip        :: a -> a
    fromRational :: Rational -> a

class  (Fractional a) => Floating a  where
    pi                  :: a
    exp, log, sqrt      :: a -> a
    (**), logBase       :: a -> a -> a
    sin, cos, tan       :: a -> a
    asin, acos, atan    :: a -> a
    sinh, cosh, tanh    :: a -> a
    asinh, acosh, atanh :: a -> a

class  (Real a, Fractional a) => RealFrac a  where
    properFraction   :: (Integral b) => a -> (b,a)
    truncate, round  :: (Integral b) => a -> b
    ceiling, floor   :: (Integral b) => a -> b

class  (RealFrac a, Floating a) => RealFloat a  where
    floatRadix          :: a -> Integer
    floatDigits         :: a -> Int
    floatRange          :: a -> (Int,Int)
    decodeFloat         :: a -> (Integer,Int)
    encodeFloat         :: Integer -> Int -> a
    exponent            :: a -> Int
    significand         :: a -> a
    scaleFloat          :: Int -> a -> a
    isNaN, isInfinite, isDenormalized, isNegativeZero, isIEEE
                        :: a -> Bool
    atan2               :: a -> a -> a
```

5.1.3 Exemplo de construção de uma Classe

Digamos que temos que construir uma classe para instanciar alguns operadores que chamaremos de `+++` e `***`, de forma que:`x***y = x * x * y`; `x+++y = 10 * x + y` se `x` e `y` forem do tipo `Integer`; e `x+++y = 2 * x + y` se `x` e `y` forem do tipo `Double`. Vamos denominar a classe de `Mf` que pode estar definida dentro de um módulo `MixFunc`.

Listagem 5.1: Módulo Mixfunc.hs

```
module Mixfunc ((+++),(***)) where

class Num a => Mf a where
  (+++) :: a -> a -> a
  (***) :: a -> a -> a

  x *** y = x * x * y

instance Mf Double where
  x +++ y = 2*x + y

instance Mf Integer where
  x +++ y = 10*x + y
```

O módulo pode ser carregado e testado no interpretador (GHCi):

```
Prelude> :l Mixfunc
[1 of 1] Compiling Mixfunc          ( Mixfunc.hs, interpreted )
Ok, modules loaded: Mixfunc.
*Mixfunc> (2::Double) *** (4::Double)
16.0
*Mixfunc> (2::Integer) *** (4::Integer)
16
*Mixfunc> (2::Integer)+++(4::Integer)
24
*Mixfunc> (2::Double)+++(4::Double)
8.0
*Mixfunc>
```

5.2 Tipos

Os tipos de dados padrões são os booleanos (`Bool`), caracteres (`Char`) e numéricos (`Int, Integer, Float` e `Double`). Além disso, a linguagem define tipos algébricos (listas e n-uplas) e tipos abstratos (funções, I/O, Maybe e outros).

Existem tipos associados, tais como o tipo `String` que é uma associação do tipo lista com o tipo caractere `[Char]`. O construtor nulo é um tipo especial de n-upla vazia (`()`).

5.2.1 Tipos de dados padrão

O tipo booleano é definido como `False | True` e é associado aos operadores `&&` (and) `||` (or) e `not`. O nome `otherwise` é considerado como verdadeiro (`True`) para legibilidade de expressões condicionais com "guardas".

O tipo caractere (`Char`) é uma seqüência de valores de 16 bits, em conformidade com o padrão Unicode. Os cacacteres de controle ASCII têm várias formas de representação (*numeric escapes, ASCII mnemonic escapes*, e a notação \^ X). Ainda, existem algumas equivalências, tais como \a e \BEL para o *beep*; \b e \BS para o *backspace*; \f e \FF para *form feed*; \r e \CR para *carriage return*; \t e \HT para tabulação horizontal; \v e \VT para tabulação vertical; e \n e \LF para *line feed*.

Em HASKELL um caractere é representado entre aspas simples, por exemplo: `'a', 'b', '\BEL'` etc.. Um tipo `String` pode ser representado entre aspas duplas (`"banana"`) ou como uma lista de caracteres, por exemplo: `['b','a','n','a','n','a']`.

O tipo `Int` são inteiros de tamanho fixo, compreendendo os valores entre -2^{29} e $(2^{29}-1)$, e o tipo `Integer` é um inteiro de precisão arbitrária. O tipo `Float` é um número real de precisão simples (32 bits) e o tipo `Double` é um real de precisão dupla (64 bits).

5.2.2 Tipos de dados algébricos

Os dados algébricos são dados relacionados com coleções de valores organizados em vetores, matrizes, pares, triplas e n-uplas ordenadas ou não ordenadas. A linguagem HASKELL tem dois tipos de dados algébricos, com os quais se pode construir todos os outros. Um deles é o tipo "lista" que pode ser uma lista numérica (vetor) uma lista de listas (matriz) ou listas de caracteres que é equivalente ao tipo `String`. Outro tipo algébrico é denominado de "tupla"

que equivale à n-upla (ênupla) em português. Uma tupla pode ser desde uma dupla, uma tripla ou conter mais de 3 elementos.

As listas são tipos algéricos formados por 2 construtores: o construtor "`:`" e o construtor "`[]`". A construção da lista é feita a partir de um ou mais elementos, encadeados com uma lista vazia como, por exemplo:

```
1:2:3:[]
```

que retorna a lista `[1,2,3]`. As listas são instâncias das classes `Read, Show, Eq, Ord, Monad` e `MonadPlus`. A definição básica deste tipo de dado é:

```
data  [a]  =  [] | a : [a]  deriving (Eq, Ord)
```

As *tuplas* são tipos de dados formados pelos construtores "`()`" e "`,`". Podem ser de qualquer tamanho, embora muitas implementações do HASKELL limitem seu tamanho em até 15 elementos. Algumas funções que lidam com tuplas (p. ex.: `zip`) são implementadas até o tamanho de 7 elementos por tupla. A construção de uma tupla pode ser escrita como `(,) x y` que resulta em `(x,y)`.

As tuplas são instâncias das classes `Eq, Ord, Bounded, Read` e `Show` e dependem dos tipos de seus elementos. Uma tupla pode ser explicitamente tipada, por exemplo: `(Int,Bool,Int)` e `(,,) Int Bool Int`.

5.2.3 Tipos abstratos

A linuagem HASKELL possui, ainda tipos abstratos de dados como funções, `Maybe, Either, Ordering` e `()`. Estes tipos serão explicitados em outros capítulos. As funções não têm construtores definidos e os outros são definidos como:

```
data  () = () deriving (Eq, Ord, Bounded, Enum, Read, Show)
ata  Maybe a     =  Nothing | Just a  deriving (Eq, Ord, Read, Show)
data  Either a b  =  Left a | Right b  deriving (Eq, Ord, Read, Show)
data  Ordering    =  LT | EQ | GT deriving
                                    (Eq, Ord, Bounded, Enum, Read, Show)
```

5.3 Exercícios

Exercício 1: Qual o tipo das funções abaixo:

```
increm = \x->x+1

ppar n = product [ 2,4 .. n]

primeiro (x:_) = x

ultimo [x] = x
ultimo (_:xs) = ultimo xs

rotaciona [] = error "Não dá para rodar: lista vazia!"
rotaciona (x:xs) = xs ++ [x]
```

Exercício 2: Dado o módulo abaixo, ache o tipo da função `triads`

Listagem 5.2: Tríades pitagóricas

```
module Pitagoras(
       triads
       ) where

triads n = [(a,b,c) |
                      a<-[1..n],
                      b<-[1..n],
                      c<-[1..n],
                      a^2+b^2==c^2]
```

Capítulo 6

Módulos

Um programa funcional pode ser composto de um programa principal (*e.g.* Main.hs) e um ou mais módulos que podem ser importados pelo programa. Um módulo tem a função de agrupar as definições que podem ser utilizadas em um programa. Um módulo é formado por um nome (com a primeira letra maiúscula), uma lista de dados a ser exportado e o corpo do módulo com as definições:

```
module Formas (...<dados exportados> ...) where

... <corpo do módulo> ...
```

Para se utilizar as definições de um módulo deve-se importá-lo no programa principal, por exemplo:

```
import Formas

main = ...
```

6.1 Construindo um módulo

Com a finalidade de compreender a construção de um módulo em Haskell , vamos utilizar um exemplo extraído do livro do Paul Hudak [9]. Podemos

construir um módulo que defina formas geométricas, definindo os tipos de dados como, por exemplo:

```
data Forma = Circulo Float
           | Quadrado Float
```

Isto significa que existem dois tipos `Forma`: um `Circulo` e um `Quadrado`, representados por um elemento `Float` e que podem ser notados por "`Circulo r`" onde r é o raio do tipo decimal e por "`Quadrado s`" onde s é o lado do quadrado em decimal. Estes dados são denominados construtores porque constroem novos valores deste tipo de dados.

Nota 6.1.1 *Evidentemente, existem muitas formas geométricas como, por exemplo, quadrados são retângulos que, por sua vez, são paralelogramos que são quadriláteros (polígonos de 4 lados). Entretanto, para efeito de estudo vamos considerar apenas algumas formas.*

Assim, vamos considerar os seguintes dados:

```
data Forma = Retangulo Float Float
           | Elipse Float Float
           | TriangRet Float Float
           | Poligono [(Float,Float)]
     deriving Show
```

A instrução **deriving Show** significa que o sistema deve retornar os valores dos tipos de dados que estão sendo definidos. Esta nova declaração define os dados `Retangulo, Elipse, TriangRet` descritos por dois elemntos de ponto flutuante e `Poligono` descrito por uma lista de pares ordenados de ponto flutuante.

Com a finalidade de facilitar a redefinição de tipos, podemos contruir estes elementos, usando a declaração **type**, como na listagem 6.1.

Listagem 6.1: Módulo Formas

```
module Formas ( Forma (Retangulo, Elipse, TriangRet,
                       Poligono
                      ),
                Raio, Lado, Vertice,
                quadrado, circulo, distEntre, area
              ) where

data Forma = Retangulo Lado Lado
           | Elipse Raio Raio
           | TriangRet Lado Lado
           | Poligono [Vertice]
     deriving Show

type Raio = Float
type Lado   = Float
type Vertice  = (Float,Float)

quadrado s = Retangulo s s
circulo r = Elipse r r

triArea :: Vertice -> Vertice -> Vertice -> Float
triArea v1 v2 v3 = let a = distEntre v1 v2
                       b = distEntre v2 v3
                       c = distEntre v3 v1
                       s = 0.5*(a+b+c)
                   in sqrt (s*(s-a)*(s-b)*(s-c))

distEntre :: Vertice -> Vertice -> Float
distEntre (x1,y1) (x2,y2) = sqrt ((x1-x2)^2 + (y1-y2)^2)

area :: Forma -> Float
area (Retangulo  s1 s2) = s1*s2
area (TriangRet s1 s2)  = s1*s2/2
area (Elipse r1 r2)     = pi*r1*r2
area (Poligono (v1:vs)) = poliArea vs
     where poliArea :: [Vertice] -> Float
           poliArea (v2:v3:vs') = triArea v1 v2 v3
                                  + poliArea (v3:vs')
           poliArea _           = 0
```

6.2 Exemplo de estratégia

Na listagem 6.1 calcula-se a área de um polígono de forma recursiva por meio de sua divisão em triângulos (veja figura 6.1). Para isso, o construtor da função `poliArea` tem que ser delimitado a, pelo menos, três parâmetros. O uso do caratere "_" (*underscore*) serve como um "coringa". Em HASKELL , quando a função tem dupla definição, o sistema verifica se a primeira se aplica, caso contrário, utiliza a outra. Neste caso, se não existir 3 parâmetros, a função retorna o valor zero.

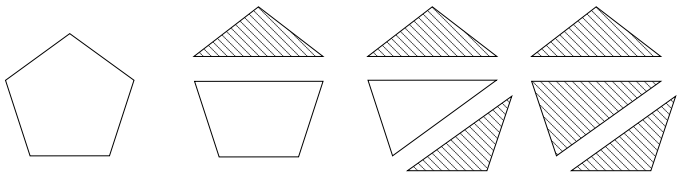
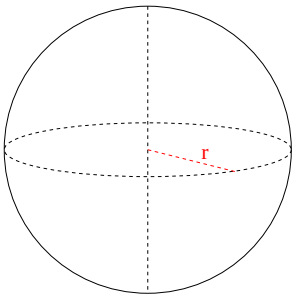


Figura 6.1: Subdivisões de um polígono em triângulos

6.3 Projeto 1

Considerando-se os sólidos geométricos abaixo e seus fatos matemáticos, escreva um módulo que possua as funções de cálculo de área e volume dos mesmos:

Esfera



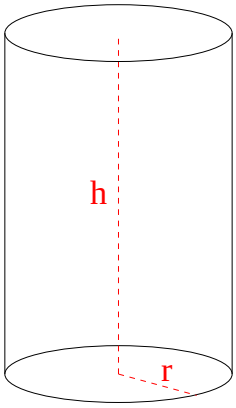
Considerando-se r = ao raio da esfera, a área da superfície é dada por:

$$A = 4\pi r^2$$

e o volume é dado por

$$V = \frac{4}{3}\pi r^3.$$

Cilindro



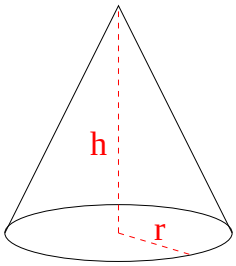
$$A_L = 2\pi r h$$

$$A_T = A_L + 2\pi r^2$$

$$V = \pi r^2 h$$

sendo A_L = Área Lateral, A_T = Área Total, V = Volume, r = raio da base e h = altura do cilindro.

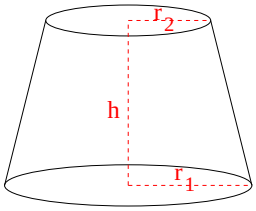
Cone



$$\begin{aligned} A_L &= \pi r\sqrt{r^2+h^2} \\ A_T &= \pi r\left(\sqrt{r^2+h^2}+r\right) \\ V &= \frac{1}{3}\pi r^2 h \end{aligned}$$

onde, A_L = Área Lateral, A_T = Área Total, V = Volume, r = raio da base e h = altura do cone.

Tronco de Cone



$$A_L = \pi(r_1+r_2)\sqrt{h^2+(r_1^2+r_2^2}$$

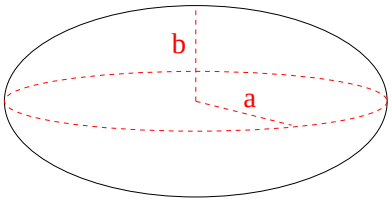
$$A_T = \pi r_1^2 + \pi r_2^2 + A_L$$

$$V = \frac{1}{3}\pi h(r_1^2+r_2^2+r_1 r_2)$$

onde, A_L = Área Lateral, A_T = Área Total, V = Volume, r_1 = raio da base, r_2 = raio da secção na altura h e h = altura.

Esferoide Oblato

Formado pela rotação da elipse sobre seu eixo menor.



$$A = 2\pi a^2 + \frac{b^2}{\epsilon} \ln\left(\frac{1+\epsilon}{1-\epsilon}\right)$$

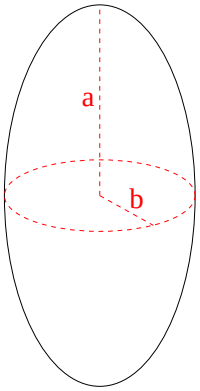
$$V = \frac{4}{3}\pi a^2 b$$

onde A = Área da Superfície, V = Volume, a = semi-eixo maior, b = semi-eixo menor, ϵ = excentricidade, calculada como:

$$\epsilon = \frac{\sqrt{a^2 + b^2}}{a}.$$

Esferoide Prolato

Formado pela rotação da elipse sobre seu eixo maior.



$$A = 2\pi b^2 + 2\pi \frac{ab}{\epsilon} \text{ arc seno}\epsilon$$

$$V = \frac{4}{3}\pi a b^2$$

onde, A = Área da Superfície, V = Volume, a = semi-eixo maior, b = semi-eixo menor e ϵ = excentricidade.

Nota 6.3.1 *Existem outras secções de cone e outros esferoides (elipsoides) que não estamos considerando aqui, bem como ângulos sólidos, paralelogramos e trapezóides. Conhecendo-se sua geometria, não será difícil implementá-los, futuramente, neste módulo.*

Capítulo 7

Mais Listas

Podemos definir uma lista como um conjunto ordenado de elementos, geralmente do mesmo tipo de dado. Por exemplo, `[20,13,42,23,54]` é uma lista numérica de inteiros; `[‘a’,‘d’,‘z’,‘m’,‘w’,‘u’]` é uma lista de caracteres; e `["banana","abacaxi","pera"]` é uma lista de frutas.

Cada linguagem tem uma sintaxe específica para organizar os dados em forma de listas. Em C/C++ uma lista pode ser descrita por uma estrutura ou uma enumeração. Por exemplo, considere uma linha de texto como sendo uma lista de caracteres:

```
 /* Neste exemplo, "struct linebuffer" é uma estrutura que
  * pode armazenar uma linha de texto. */

struct linebuffer{
  long size;
char *buffer;
};
```

Nas linguagens funcionais, a declaração de listas usa os construtores `[ ]` e `:` , onde `[]` representa uma lista vazia. Assim, uma lista pode ser declarada como `[1,2,3]` ou `(1:2:3:[])`. Além disso, pode-se declarar uma lista usando-se uma expressão conhecida como *list comprehension*:

```
[x | x <- [1,2,3]]
```

Estas expressões derivam do cálculo-λ e podem ser escritas como: `(\x -> x) [1,2,3]` que, em cálculo-λ é escrita como:

$$\lambda x.x\{1,2,3\}$$

Liguagens como Haskell , Clean, CAML e Erlang, por exemplo, suportam tanto declarações explícitas de listas, tais como `[1,2,3,4,5]` quanto implícitas, ou seja, `[1..5]`. Este último caso representa uma progressão aritmética (PA) de razão 1. Assim, por exemplo:

```
[2,4..20]     -- representa [2,4,6,8,10,12,14,,16,18,20] ou
              -- uma PA de 2 a 20 de razão 2
[1,4..15]     -- representa [1,4,7,10,13], ou uma PA de 1 a
              -- 15 de razão 3.
```

As listas implícitas podem ser declaradas de forma finita (`[1..20]`, `[2,4..100]`, `['a'..'m']` etc.) ou de forma infinita: (`[1..]`, `[2,4..]` etc.).

Existe, ainda, uma maneira de representar uma lista por `(x:xs)`[1] que é uma meta-estrutura onde x representa o primeiro elemento da lista (chamado de "cabeça") e xs representa o restante da lista (chamado de "cauda").

7.1 Listas e operações

Uma vez que uma das principais estruturas podem ser organizadas em listas, vamos explorar um pouco mais o conceito, tipagem e construção de listas em Haskell .

Agora que já conhecemos um pouco mais sobre tipos e listas, é importante saber que o Haskell (e outras linguagens funcionais como o Clean, Caml e Erlang) apresenta algumas funções definidas para a manipulação de dados de uma lista. Mas, o que há de tanta importância em listas. Bem, primeiramente, já vimos que uma cadeia de caracteres, nada mais é do que uma lista, depois, grande parte das operações matemáticas e tratamento de dados são feitos sobre listas, por exemplo, dados de uma tabela de banco de dados, vetores são listas, matrizes são listas de listas etc.. Se considerarmos que endereços de memória, set de instruções de processadores, fluxo de dados etc., também podem ser tratados como listas, justificamos porque lidar com listas é tão importante.

[1] Lê-se 'xis' e 'xises'.

As linguagens funcionais implementam algumas funções para operações sobre dados de uma lista. Por exemplo, a função `map`, em `map f L` aplica uma função `f` em uma lista `L`:

```
Prelude> map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
Prelude> map odd [1..10]
[True,False,True,False,True,False,True,False,True,False]
```

a função `filter` testa uma condição em uma lista:

```
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> filter (>10) [1..20]
[11,12,13,14,15,16,17,18,19,20]
Prelude> filter (>5) (filter (<10) [1..20])
[6,7,8,9]
```

Na tabela 7.1, encontram-se descritas algumas funções.

A linguagem HASKELL já tem muitas funções implementadas no módulo `Prelude` e cujos tipos podem ser verificados no interpretador, por meio do comando `:info <função>`. Alguns exemplos são:

- A função `concat` é definida no módulo `Prelude` como uma função que recebe uma lista de listas e retorna uma lista concatenada das listas do argumento. A função é definida em função da função `foldr` (*fold right*).

  ```
  concat :: [[a]] -> [a]
  concat xss = foldr (++) [] xss
  ```

  onde, a função `foldr` toma o segundo argumento e o último elemento da lista e aplica uma função, tomando o penúltimo item da lista e aplica o resultado e assim por diante.

  ```
  foldr            :: (a -> b -> b) -> b -> [a] -> b
  foldr f z []     =  z
  foldr f z (x:xs) =  f x (foldr f z xs)
  ```

Função	Descrição	Sintaxe	Exemplo
`concat`	Concatena duas listas	`concat [[<lista 1>],[<lista 2>]]`	`concat [[1..3],[6..8]]` Resultado: `[1,2,3,6,7,8]`
`filter`	Aplica uma condição e retorna os elementos que casam com o padrão	`filter <padrão> [<lista>]`	`filter odd [1..10]` Resultado: `[1,3,5,7,9]`
`map`	Aplica uma função em todos os elementos de uma lista	`map <função> [<lista>]`	`map (*2) [1,2,3,4]` Resultado: `[2,4,6,8]`
`take`	Toma n elementos de uma lista	`take n [<lista>]`	`take 5 [10,20..]` Resultado: `[10,20,30,40,50]`
`zip`	Forma duplas com os elementos de duas listas	`zip [<lista 1>] [<lista 2>]`	`zip [1,2,3] ['a','b','c']` Resultado: `[(1,'a'),(2,'b'),(3,'c')]`

Tabela 7.1: Exemplos de funções para operações sobre listas em HASKELL .

- A função `filter` testa um elemento de uma lista com o padrão e o retorna caso seja verdadeiro. Ela é definida como:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs
```

- A função `map` aplica uma função a todos elementos de uma lista, sendo definida no módulo `Prelude` como:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

- A função `take` toma o número de elementos do primeiro argumento na lista passada como segundo argumento. Sua definição é:

```
take                    :: Int -> [a] -> [a]
```

```
take n _      | n <= 0 =  []
take _ []             =  []
take n (x:xs)         =  x : take (n-1) xs
```

As operações sobre listas podem ser definidas em funções do usuário. Por exemplo, dada uma lista de elementos finitos, podemos gerar todas as combinações possíveis destes elementos. A listagem 7.1 mostra um exemplo de programa de permutações.

Listagem 7.1: Programa Permuta.hs para explicar operações sobre listas em Haskell.

```
module Permuta (
        perms
        ) where
perms :: [a] -> [[a]]
perms []    = [[]]
perms (x:xs)    = concat (map (inter x) (perms xs))

inter x []              = [[x]]
inter x ys'@(y:ys)      = (x:ys') : map (y:) (inter x ys)
```

Nota 7.1.1 *O "@" que é chamado de aliás (as-pattern). Estes aparecem sempre como* `<identificador>@<padrão>`*, onde o identificador representa todo o objeto e o padrão o decompõe em seus componentes. É importante ressaltar que os aliases "@" têm precedência sobre o construtor ":" e* `a:b@c:d` *vai associar* `b` *com* `c` *e não com* `c:d`*. Quando aplicada a expressão* `a@(b:c@(d:e))` *no argumento* `[1,2,3,4]`*, teremos* `a = [1,2,3,4]`*,* `b = 1`*,* `c = [2,3,4]`*,* `d = 2` *e* `e = [3,4]`*.*

O módulo "Permuta"retorna todas as listas possíveis a partir de uma lista finita, por exemplo:

```
Prelude> :l Permuta
Permuta> perms "abcd"
["abcd","bacd","bcad","bcda","acbd","cabd","cbad","cbda","acdb",
"cadb","cdab","cdba","abdc","badc","bdac","bdca","adbc","dabc",
"dbac","dbca","adcb","dacb","dcab","dcba"]
```

```
Permuta> perms [1..4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1],[1,3,2,4],[3,1,2,4],
[3,2,1,4],[3,2,4,1],[1,3,4,2],[3,1,4,2],[3,4,1,2],[3,4,2,1],
[1,2,4,3],[2,1,4,3],[2,4,1,3],[2,4,3,1],[1,4,2,3],[4,1,2,3],
[4,2,1,3],[4,2,3,1],[1,4,3,2],[4,1,3,2],[4,3,1,2],[4,3,2,1]]
```

A melhor maneira de compreender as funções de uma linguagem é usando-as. Então, vamos dar alguns exemplos de funções que operam com listas. Digamos que a partir de uma função qualquer, queiramos que ela seja aplicada a toda a lista. Já vimos que para isso podemos usar a função `map`. Por exemplo:

Listagem 7.2: Exemplo do código do mapeamento de uma função

```
main = print(map quad [1..10])

quad :: Int -> Int
quad n = n*n
```

Como já vimos, a função `map` "mapeia" uma função em todo o argumento, ou seja, uma lista. Pode-se testar esta função no interpretador, por exemplo:

```
Prelude> let quad x = x*x
Prelude> map quad [1..10]
[1,4,9,16,25,36,49,64,81,100]
Prelude> map quad (filter even [1..10])
[4,16,36,64,100]
Prelude>
```

O trabalho com listas é bastante prático para lidar com dados. Por isso vamos expor mais um exemplo. Digamos que seja necessário eliminar elementos adjacentes em duplicata, dentre os elementos de uma lista. Então, podemos criar uma função, que chamaremos de `remdup`, para resolver o problema:

Listagem 7.3: Função para remover dados duplicados em uma lista.

```
module RemDup (remdup) where

remdup :: [Int] -> [Int]
remdup [] = []
remdup (x:xs@(y:_)) | x == y    = remdup xs
                    | otherwise = x:(remdup xs)
remdup xs = xs
```

Para compreender o funcionamento deste programa, carregue-o no interpretador e experimente:

```
*Main> remdup [1,2,2,3,4,4,5]
[1,2,3,4,5]
*Main>
```

Agora observe o que acontece se a lista for uma lista de caracteres:

```
*Main> remdup "Jooao"
:1:
     Couldn't match 'Int' against 'Char'
          Expected type: [Int]
          Inferred type: [Char]
          In the first argument of 'remdup', namely '"Jooao"'
          In the definition of 'it': remdup "Jooao"
*Main>
```

Isto acontece porque a função `remdup` foi tipada como `Int` (veja na listagem 7.3). Como `remdup` foi declarada como uma função que recebe uma lista de inteiros e retorna uma lista de inteiros, ao tentarmos aplicar a função a uma lista de caracteres o programa avisa que não pode "casar" o tipo inteiro com o tipo caractere. Este é um problema típico de estrutura de dados, pois as operações sobre um tipo é limitada pela tipagem dos dados sobre os quais operam.

7.2 Polimorfismo

Para resolver o problema do programa anterior (listagem 7.3) de forma que a função seja aplicável a uma lista genérica de qualquer tipo, vamos introduzir o conceito de **polimorfismo**.

Quando a tipagem de uma função tem mesmo tipos de entrada e de saída, dizemos que a função é monomórfica, caso contrário dizemos que a função é polimórfica. Se precisamos de uma função que vai operar sobre uma lista de caracteres ou uma lista de inteiros. É claro que podemos definir uma função para cada tipo de lista (`[Int]` ou `[Char]`), mas podemos especificar uma lista genérica como `[a]` (`a` como em *any*, em ingles). Esta forma de tipagem é denominada "polimorfismo".

Dizemos que uma variável é polimórfica quando aceita qualquer tipo de argumento. Em linguagem funcional usa-se a letra "a" (de *any*) para nos referirmos a este tipo de variável. Então, para que nosso programa funcione para qualquer tipo de lista podemos modificá-lo como na listagem 7.4.

Listagem 7.4: Função para remover dados duplicados em uma lista de qualquer tipo.

```
module RemDupAny (remdupany) where

remdupany :: (Eq a) => [a] -> [a]
remdupany [] = []
remdupany (x:xs@(y:_)) | x == y    = remdupany xs
                       | otherwise = x:(remdupany xs)
remdupany xs = xs
```

Agora o programa pode tratar qualquer tipo de lista. Carregue-o no interpretador e experimente com todos os tipos de lista.

7.3 Classes e Polimorfismo

Note que para definirmos a função usamos uma classe "Eq" que é definida como:

```
class Eq a where {
```

```
(/=) :: a -> a -> Bool {- has default method -};
(==) :: a -> a -> Bool {- has default method -};
}
```

Elementos em uma classe são ditos estarem em um contexto. Neste caso, restringimos que as variáveis polimórficas "a" são válidas no contexto onde podem ser testadas pela equalidade (igual `(==)` ou diferente `(/=)`). Tal procedimento é necessário, pois existem casos em que definir `[a] -> [a]` não permitem outras comparações como "menor que" (`<`) ou "menor ou igual a" (`<=`). Na prática, nem todos os objetos estão sujeitos a um teste de inequalidade.

As classes que podem ser utilizadas em declarações com atributos polimórficos são:

- **Eq a** - Restringe o uso em objetos que permitem ser testados quanto à equalidade (== e /=).
- **Ord a** - Restringe o uso em objetos que podem ser testados quanto à ordenação (<, <=, > e >=).
- **Num a** - Esta classe define que as funções podem ser aplicadas a objetos definidos como numéricos (`Int, Float, Real, Complex, Ratio` etc.).

7.4 Projeto 2

Parte 1 - Séries finitas

A soma dos números de uma lista de inteiros pode ser dada por:

$$\sum_{1}^{n} \frac{n(n+1)}{2}$$

onde n é o último termo da série. A lista das somas pode ser implementada como:

```
somaInt n = map (\x->(x*(x+1)/2)) [1..n])
```

e, evidentemente, a soma de uma lista pode ser calculada como:

```
totaLista n = last somaInt n
```

Ache o mesmo para uma lista de números ímpares, pares, quadrados dos inteiros e quadrados dos ímpares.

Parte 2 - Séries infinitas

O número inteiro 1 pode ser obtido pela soma dos termos da série infinita

$$\sum_{n=1}^{\infty} \frac{1}{n(n+1)}$$

e pode ser implementada uma função `quaseUm` como:

```
quaseUm n = sum (map (\x->(1/(x^2+x))) (take n [1..]))
```

Implemente as funções: `quaseDois` para o inteiro 2; e `quaseE` para o exponencial e, ou base dos logaritmos naturais[2].

[2]O número e pode ser calculado por uma série infinita:

$$1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!} + \ldots = 2.7182818285$$

.

Capítulo 8

Mônadas e Homologia

O conceito de Mônadas é fundamental para a compreensão de como se pode implementar um paradigma procedural de forma funcional. Aparentemente, pode parecer um paradoxo, já que a filosofia do paradigma funcional é utilizar funções de forma declarativa e não na forma de procedimentos imperativos. No entanto, algumas ações computacionais são imperativas por natureza, pois imprimir, ler de arquivos ou de entrada padrão e escrever para arquivos ou saída padrão requer comandos do tipo `print, read` e `write`.

A maneira de se realizar um fluxo procedimental de instruções de forma declarativa está relacionado com um conceito matemático de mônadas e para compreender este conceito, é necessário que se compreenda alguns tópicos de homologia ou álgebra homológica.

8.1 Álgebra homológica

A álgebra homológica é um ramo relativamente recente da matemática, que estuda homologia de funções, dentro da teoria das categorias (topologia).

Uma categoria pode ser descrita como uma estrutura matemática abstrata que guarda relações entre si. Por exemplo, dadas as funções:

$$\begin{aligned} f : x &\rightarrow y \\ g : y &\rightarrow z \end{aligned}$$

existe uma função $h : x \rightarrow z$ que pode ser um functor $F(f \circ g) = F(f) \circ F(g)$. Se existir uma relação, tal que:

$$H(f(x), y) \equiv H(x, g(y))$$

dizemos que existe uma relação homóloga de "equivalência natural"entre f(x) e g(y). Isto significa que existe um functor H que faz o mapeamento de y em $f(x)$ e de x em $g(y)$. Este functor estabelece um tipo especial de mapeamento entre categorias (morfismos). Por exemplo:

Sejam C e D duas categorias algébricas (espaços topológicos), um functor F de C para D é um mapeamento tal que:

1. associa a cada objeto $x \in C$ um objeto $F(x) \in D$,
2. associa a cada morfismo $f : x \to y \in C$ um morfismo $F(f) : F(x) \to F(y) \in D$.

satisfazendo as condições:

1. de identidade: $F(id\, x) = id\, F(x)$ para todo objeto $x \in C$; e
2. de associatividade (composição): $F(g \circ f) = F(g) \circ F(f)$ para todos os morfismos $f : x \to y$ e $g : y \to z$.

ou seja: functores devem preservar morfismos de indentidade e de composição.

8.2 Mônadas e categorias

Seja C uma categoria, uma mônada em C consiste de um functor:

$$T : C \to C$$

e duas transformações naturais:

$$\begin{aligned} \eta : 1C &\to T \\ \mu : T^2 &\to T \end{aligned}$$

onde $1C$ é o functor identidade em C e T^2 é o functor $T \circ T$ de C para C, tal que satisfaçam as seguintes condições (chamadas de condições de coerência):

1. 1. $\mu \circ T\mu = \mu \circ \mu T$ são transformações naturais $T3 \to T$ (veja fig. 8.1).
2. $\mu \circ T\eta = \mu \circ \eta T = 1T$ são transformações naturais $T \to T$ onde 1T é a transformação identidade de T para T (veja fig. 8.2).

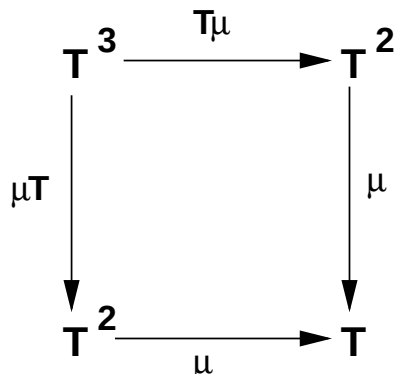


Figura 8.1: Diagrama comutativo $T^3 \to T$

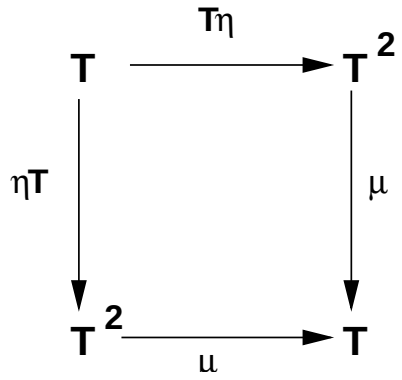


Figura 8.2: Diagrama comutativo $T \to T$

Assim, se

$$H(F(x), y) \equiv H(x, G(y))$$

e

$$T = F(x) \circ G(y)$$

de forma que F e G sejam functores levo-associativo de F para G, a composição T é denominada uma "Mônada".

8.3 Mônadas e linguagem funcional

Na prática, o uso de mônadas, em uma linguagem funcional, serve para estruturar operações que devem ser executadas em uma determinada ordem (uma operação "funcional imperativa").

Por exemplo, observe o problema do cálculo da resistência de dois resistores em paralelo. Sabemos que a resistência é calculada pela equação:

$$\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} \tag{8.1}$$

A função pode ser implementada da seguinte maneira:

```
-- rst: função que toma 2 floats e retorna outro.

 rst :: Float -> Float -> Float
 rst x y = 1 / ((1 / x) + (1 / y))
```

Entretanto, se qualquer resistor for de 0 Ohms, a função deverá tratar a divisão por zero. Pode-se, então, criar um operador de divisão que trate o problema internamente:

```
-- // é um operador de divisão que toma dois "Maybe Float"s e
-- retorna outro.
-- "Maybe Float" é um tipo extendido de Float que representa
-- resultados que podem não ocorrer.

(//) :: Maybe Float -> Maybe Float -> Maybe Float
x // y = ...   -- definiremos depois

-- Apenas para exemplo. Existe um erro pois os tipos não combinam
par :: Float -> Float -> Maybe Float
par x y = 1 // ((1 // x) + (1 // y))
```

A definição do operador "//" pode ser descrita como:

```
x // y = do
a <- x
b <- y
if b == 0 then Nothing else Just (a / b)
```

onde `Nothing` e `Just` são valores das mônadas que podem ser "nada" ou "apenas" opera a função. Desta forma, pode=se implementar corretamente a função `rst` de forma monádica:

```
add x y = do
x' <- x
y' <- y
  return (x' + y')
rst x y = let
one = return 1
rx = return x
ry = return y
in one // (add (one // rx) (one // ry))
```

O módulo completo com a função implementada em forma de mônada pode ser visto na listagem 8.1.

Listagem 8.1: Módulo Resist.hs

```
module Resist (rst) where

x // y = do
    -- extrai os valores de x e y:
    a <- x
    b <- y
    if b == 0 then Nothing else Just (a / b)

soma x y = do
    x' <- x
    y' <- y
    return (x' + y')

-- definição monádica da operação:
rst x y = let
    one = return 1
    rx = return x
    ry = return y
    in one // (add (one // rx) (one // ry))
```

Capítulo 9

E/S (IO)

A maioria dos programas têm que estabelecer um diálogo ou uma interface com o usuário, com outros programas e/ou arquivos. Para que isto seja possível, estas ações de "entrada" e "saída" (E/S) de dados dependem do tipo de dados que o programa recebe para tratar e que tipos de dados o programa devolve. Em linguagem funcional, a entrada de dados está, geralmente, ligada aos objetos (argumentos) a serem tratados pelas funções. A saída é o resultado desta computação.

Hudak [8] chama a atenção sobre a diferença entre o sistema de E/S das linguagens procedurais (imperativas) e as funcionais (declarativas). Para ele, o sistema de E/S é tão robusto, apesar de funcional, quanto nas liguagens tradicionais de programação. Nas linguagens de programação imperativas o processo é feito por ações e modificações de estado por meio da leitura, alteração de variáveis globais, escrita de arquivos, leituras a partir do terminal e criação de "janelas". Estas ações também podem ser levadas a cabo em um estilo funcional mas, são claramente separadas do cerne funcional da linguagem.

Para lidar com o problema de E/S, a linguagem funcional usa um conceito matemático de **Mônadas**[1], mas o conhecimento estrito do conceito de mônada não é necessário para que possamos compreender como funciona o processo Por enquanto, vamos considerar a **classe IO ()** como um tipo abstrato de dado[2].

[1]Mônadas são construtores de tipos usados com o paradigma de computação sequencial (imperativa), implementada de forma funcional (declarativa).

[2]A linguagem Clean utiliza o conceito de "tipo único", no lugar de mônadas, para lidar com o problema de E/S.

Listagem 9.1: Exemplo de IO em linguagem funcional.

```
import IO (getChar)
main :: IO ()
main = do
   putStrLn "Entre 's' ou 'n'e tecle <enter>:"
   carct <- getChar
   let resp | carct == 's'   = "sim"
            | carct == 'n'   = "não"
            | otherwise  = "nem sim, nem não"
   putStrLn resp
```

9.1 Operações de IO

O melhor jeito de compreender as ações de E/S de um sistema é observando, na prática como tudo acontece. Vamos, então, observar como podemos passar um caractere para um programa em Haskell .

Use o compilador Haskell (GHC):

```
ghc -o <nome_do_executável> <nome>.hs
```

No exemplo da listagem 9.1, começamos importanto uma função `getChar` de um módulo chamado "IO". Na verdade, o programa funcionaria sem isto, uma vez que esta função já se encontra definida no "Prelude". Entretanto, estamos fazendo isso apenas para demonstrar como se importa algo de um módulo. Esta declaração pode ser comparada ao `#include` do C. Em seguida declaramos que o main é do tipo `IO ()`. O restante do programa é bastante legível. A função `getChar` espera um caractere do teclado para o terminal e o passa para a variável `carct`. Neste caso ela toma apenas um caractere, sendo que se digitarmos mais de um caractere, apenas o primeiro será aceito. A função `putStrLn` imprime uma linha (incluindo
n) no terminal.

Outro tipo de IO seria, por exemplo, o de comunicação entre o programa e arquivos. Para isso, podemos programar um aplicativo que conta as linhas, as

Listagem 9.2: Exemplo de IO com arquivos.

```
import System (getArgs)
main :: IO ()
main = do
   args <- getArgs
   case args of
       [fname] -> do fstr <- readFile fname
             let nWords = length . words $ fstr
                 nLines = length . lines $ fstr
                 nChars = length fstr
             putStrLn "lin\tpal\tcar\tarq"
             putStrLn . unwords $ [ show nLines, "\t"
                                  , show nWords, "\t"
                                  , show nChars, "\t"
                                  , fname]
       _ -> putStrLn "uso: meu-wc <nome_do_arquivo>"
```

palavras e os caracteres de um arquivo texto que chamaremos de "meu-wc.hs" (algo semelhante com o comando wc do unix).

Na listagem 9.2, o tipo "$"é um operador de baixa precedência à direita e "." é um operador de alta precedência à direita. Isto significa que a expressão:

```
nWords = length . words $ fstr
```

é equivalente à `length (words (fstr))`.

9.2 Manipulando arquivos

A manipulação de arquivos em Haskell ou outra linguagem funcional não difere muito das liguagens tradicionais. Por exemplo, se quisermos copiar um arquivo para outro é preciso usar um manipulador (*handle*) das operações

Listagem 9.3: Exemplo de manipulação de arquivos.

```
import IO

main :: IO ()

main = do
   fromHandle <- getAndOpenFile "Copiar de: " ReadMode
   toHandle   <- getAndOpenFile "Copiar para: " WriteMode
   contents   <- hGetContents fromHandle
   hPutStr toHandle contents
   hClose toHandle
   putStrLn "Feito."

getAndOpenFile          :: String -> IOMode -> IO Handle

getAndOpenFile prompt mode =
   do putStrLn prompt
     name <- getLine
     catch (openFile name mode)
           (_ -> do putStrLn ("Impossível abrir "++ name)
                    getAndOpenFile prompt mode)
```

de E/S. Quando importamos o módulo IO várias classes e tipos podem ser utilizados e que não estão presentes no módulo padrão (Prelude).

Vejamos então um exemplo prático que podemos chamar de "meu-cp.hs", por ser uma variante do "cp" do unix.

Note que quando abrimos um arquivo, é criado um manipulador do tipo `Handle` para as transações de E/S (listagem 9.3). Ao fecharmos o manipulador associado ao arquivo, fechamos o arquivo. Observe também que `IOMode` é uma classe definida como:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

e `openFile` é descrito como:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Um ponto interessante a ser comentado é que, como a função `getContents` usa um *handle* de E/S do tipo `String`:

```
getContents :: IO String
```

Por isso, pode parecer que a função lê um arquivo ou uma entrada inteira e que resulta em uma performance comprometida em certas condições de espaço de memória e tempo de processamento. Aí está uma grande vantagem da linguagem funcional. Como uma *string* é uma lista de caracteres, a função pode retornar uma lista sob demanda devido ao caracter não-estrito da avaliação (*lazy evaluation*).

Capítulo 10

Programas

Quando se trata de programação em geral, tornar os programas modulares permite o rápido e eficiente reaproveitamento do código. Normalmente, em C/C++ a criação de bibliotecas (*libs*), cabeçalhos (*header files*), funções e classes são responsáveis pela modularidade e orientação ao objeto (OOP). Nas liguagens funcionais, as funções, classes e tipagens (*types*) podem ser agrupadas ou divididas em módulos. Estes módulos servem para, principalmente, controlar o domínio das funções e abstrações de tipagem dos dados.

Um módulo em HASKELL , por exemplo, contém várias declarações que já vimos. Geralmente, encontramos declarações de delimitação de domínio, declarações de tipagem e dados, declarações de classes e instâncias, definições de funções etc.. Exceto pelo fato de que as declarações a serem importadas de outros módulos, devam ser declaradas no início de um módulo, as declarações do módulo podem aparecer em qualquer ordem.

10.1 Um exemplo prático

Vamos partir de um exemplo para entender como montar um programa. Para isso, vamos partir de um problema simples: a conversão de um arquivo de áudio do tipo Riff/Wave (.wav) de 8 bits, mono para um arquivo de dados numéricos.

Nota 10.1.1 *Este tipo de arquivo é formado por 44 bytes de cabeçalho, seguido do segmento de dados, onde cada byte (8 bits) representa um dado valor. O programa deve, então separar os primeiros 44 bytes e ler cada byte seguinte convertendo-o para um valor numérico inteiro entre* -128 *até* 128

A listagem 10.1 apresenta um módulo (`Converte.hs`) que exporta duas funções: `leNum` e `wavDat8`:

Listagem 10.1: Módulo Converte.hs

```
module Converte (leNum, wavDat8) where

import IO

wavDat8 :: Int -> Int
wavDat8 x | x == 128  = 0
          | otherwise = x - 128

leNum :: IO Integer
leNum =  do { line <- getLine; readIO line }
```

O programa principal executa as operações de E/S, ou seja, toma os parâmetros de comando, lê o arquivo Riff/Wave e grava o arquivo de dados (veja listagem 10.2).

Listagem 10.2: Módulo Main.hs

```
import System (getArgs)
import IO
import Char
import Converte

main :: IO () -- Módulo principal do programa

main = do
  args <- getArgs
  case args of
     [fname] -> do
        fstr <- readFile fname
        arqSaida <- openFile "wavdat8.dat" WriteMode
        let dadosAsc = snd (splitAt 44 (map ord fstr))
        let dados    = map wavDat8 dadosAsc
        hPrint arqSaida (dados)
        hClose arqSaida
     _       -> putStrLn "uso: wc fname"
```

O programa da listagem 10.2 lê um arquivo .wav de 8 bits e converte os dados de áudio uma lista de inteiros de 8 bits. A lista resultante tem o formato `[Int]` que é ideal para ser utilizada em outros programas (filtros, transf. de Fourier etc.)

Para separar a lista de inteiros geradas em uma lista de dígitos do tipo `["2","0","-3", ...]`, use algo como:

```
let result   = map show dados
hPrint arqSaida (result)
```

Se preferir uma lista separada como `["2\n0\n-3\n ..."]` use:

```
hPrint arqSaida (unlines result)
```

Para obter uma lista contendo apenas os dígitos do tipo "2 0 -3 ..." use:

```
hPrint arqSaida (unwords result)
```

O módulo importador herda as declarações do módulo importado. Entretanto, em alguns casos, é interessante importar apenas algumas declarações de um módulo. Seja por razões de economia de código, seja para evitar redefinições de outras funções que podem ser sobrescritas durante a importação[1]. Neste caso, deve-se optar pela importação apenas da declaração que interessa, como já vimos em um exemplo da listagem 9.1:

```
import IO (getChar)
```

Neste caso, apenas a declaração de `getChar` é exportada pelo módulo IO para o programa que o está importando.

Por meio da construção de módulos, pode-se definir tipos abstratos de dados ou *Abstract Data Types* (ADT). Nestes casos, o módulo exporta apenas um tipo *hidden* de representação e todas as operações são levadas a termo em um nível de abstração que não depende de como o objeto é representado.

[1]Existem opções de alerta do compilador para evitar conflitos e substituições de declarações. Além disso, existem maneiras de se evitar o conflito explícito de entidades com o mesmo nome.

10.2 Principais aspectos dos módulos

Ao utilizar os módulos, devemos levar sempre em consideração. os seguintes aspectos:

- Uma declaração `import` pode ser seletiva, por meio do uso de cláusulas `hiding` que permitem a exclusão explícita de entidades ou objetos.
- A declaração `import` pode usar a cláusula as quando for conveniente usar um qualificador diferente do nome do módulo importado.
- Os programas importam, implicitamente por default, o módulo `Prelude`. Se for necessário excluir algum nome durante a importação, é necessário explicitá-la. Por exemplo:

  ```
  import Prelude hiding lenght
  ```

  não importará a função `lenght`, permitindo que se possa redefiní-la.
- Instâncias (*instances*) não são especificadas na lista de importação ou exportação de um módulo. Normalmente, todos os módulos exportam as declarações de `instance` e cada importação traz essas declarações para o escôpo do programa.
- As classes podem ser declaradas com construtores, entre parênteses contendo o nome da classe e suas variáveis.

10.3 Nomes qualificados

Um problema óbvio é o fato de poder acontecer de diferentes entidades ter o mesmo nome. Para que se possa evitar este problema é necessário importar usando o atributo `qualified` para a entidade cujo nome se deseja qualificar. O nome importado receberá, como prefixo, o nome do módulo importado seguido de ponto (“.”). Assim, por exemplo:

```
import qualified IO (getChar)
```

Neste caso, o nome `getChar`, será referenciado por `IO.getChar`[2].

[2]Note que x.y é diferente de x . y. O primeiro é um nome qualificado enquanto o segundo é uma função “.” (infix).

10.4 Projeto final

Como projeto final, o aluno deve implementar um programa que leia um arquivo de imagem do tipo RGB, de 24 bits, separe as camadas e converta cada uma para um arquivo de dados numéricos com ponto flutuante no formato texto.

Referências

[1] Abramsky, S. "The lazy lambda-calculus." *In*: Turner, D. A. (ed.), **Research Topics in Functional Programming**, Year of Programming series, Addison-Wesley, 1990.

[2] Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM*, 21, 613-641, 1978.

[3] Barwise, J. "Mathematical proofs of computer correctness", *Notices of the American Mathematical Society*, 7, 844-851, 1989.

[4] Church, A. "An Unsolvable Problem of Elementary Number Theory", *Am. J. Math.*, 58: 345–363, 1936.

[5] Curry, H. B.; Feys, R.; e Craig, W. "Combinatory Logic", *in:* **Studies in Logic and the Foundations of Mathematics**, Vol. I, Amsterdan: North-Holland, 1958.

[6] Eyler, P. "The Rise of Functional Language", *Linux Journal*, april, 2007.

[7] Hudak, P. "Conception, evolution, and application of functional programming languages", *ACM Computing Surveys*, 21, 359-411, 1989.

[8] Hudak, P. e Fasel, J. H. "A Gentle Introduction to Haskell", *Tech. Ret,*, Department of Computer Science, Yale University, 1992 [`http://cs-www.cs.yale.edu/homes/hudak-paul/`]

[9] Hudak, P. **The Haskell School of Expression: Learning Functional Programming through Multimedia**, Cambridge University Press, New York, 2000.

[10] McCarthy, J. "History of Lisp", *In*: **History of Programming Languages Conference**, *ACM SIGPLAN*, 173–196, june, 1978.

[11] MacLennan, B. J. **Functional Programming — Practice and Theory**, Reading, Massachusetts: Addison-Wesley Pub. Co., 1990.

[12] Paulson, L. C. "A higher-order implementation of rewriting", *Science of Computer Programming*, 3, 119-149, 1983.

[13] Raphael, B. "The structure of programming languages", *Communications of the ACM*, 9, 155-156, 1966.

[14] Schönfinkel, M. "Über die Bausteine der mathematischen Logik", *Mathematische Annalen*, 92: 305–316, 1924.

[15] Spinellis, D. "Implementing Haskell: Language Implementation as a Tool Building Exercise", *Software: Concepts & Tools*, 14:3748, 1993.

[16] Thompson, S. **The Craft of Functional Programming**, Addison-Wesley: England, 2nd. Ed., 1999.