Haskell, uma linguagem de programação funcional

Praciano-Pereira, Tarcisio *

24 de dezembro de 2019
preprints da Sobral Matemática
no. 2019.10
Editor Tarcisio Praciano-Pereira
tarcisio@sobralmatematica.org

Resumo

Estou descrevendo uma introdução à linguagem de programação Haskell, na verdade escrevendo para aprender. Existem algumas linguagens de programação que são bem teóricas do ponto de vista da ciência da Computação, entre estas se encontram os diversos dialetos de `LISP, Haskell e Forth` que se destacam de todas as outras que formam o grande padrão `C++, C, Java, Python`. `Haskell e Forth` são ditas puramente funcionais. `LISP` é um caso a parte, foi uma invenção dum matemático John McCarthy, é uma linguagem matemática.

palavras chave: Haskell, Forth, LISP

This is a very introduction to `Haskell` programming language sometimes mixing with `LISP`. I have discovered `Haskell` accidentally in a discussion about `LISP`, I got curious with its very mathematical background and decided myself to learn it, this is the reason of this paper. I will do sometimes a comparison with `LISP`, but they are quite different though, perhaps, with a similar intent to produce a deeply logic programming language. `LISP` was the invention, initially, of the mathematician John McCarthy. I will sometimes put `Forth` in line as it is quite similar to `Haskell`.

keywords: Haskell, Forth, LISP

*tarcisio@sobralmatematica.org

Este artigo ainda está sendo redigido e quando atingir a sua versão final, esta observação irá desaparecer. E porque publicar uma *versão em produção*? Porque esta página é de préprints portanto contém trabalhos com os quais os autores almejam uma publicação futura e nos quais os autores se expõem na esperança de encontrar uma colaboração.

Uma outra razão desta observação inicial é de organização da página, estou neste momento apenas reservando um número de publicação, um aviso para os que visitarem a página que este artigo está sendo escrito. Quando pronto, este aviso desaparecerá.

1 Programação funcional

haskell é uma linguagem *puramente funcional*. Isto quer dizer que todos os *símbolos* da linguagem representam uma função, isto é, estão associados a um algoritmo que transforma um símbolo que ele recebe em outro *símbolo* que deve ser uma função.

É óbvio que a frase assim contém uma *falha lógica*, na verdade os *símbolos* de `haskell` estão associados internamente a algoritmos e o que eu deveria ter dito é que um *algoritmo* recebe outro *algoritmo* identificado por um *símbolo*.

`haskell` é uma linguagem desenvolvida por professores duma universidade americana e é distribuída sob `GPL` que você pode instalar nas distribuições `GNU/Linux` com o pacote de nome `ghci` que provavelmente são as iniciais de `Glasgow haskell compiler`. Com este pacote instalado há possivelmente diversas formas de usar a linguagem, e eu vou descrever a minha favorita. Como eu uso `Linux` eu tenho hábito de usar `terminal` onde executo comandos, mas você pode fazer tudo que eu disser usando os `menus` para selecionar os programas. Faça as coisas do seu jeito!

`ghci` é o Glasgow haskell compiler, interpretado.

Abra um `terminal` e digite `ghci` e você vai se encontrar num terminal operando o *interpretador* do `haskell`. Se não tiver instalado, os sistemas `Debian`, como `Debian/GNU/Linux`, `Mint` ou `ubuntu` devem lhe apresentar uma mensagem mostrando como instalar. Mais abaixo eu lhe informo como fazer.

Interpretador ? é que as linguagens de programação se dividem em dois grandes grupos, entre outros, *interpretadas* e *compiladas*. A diferença entre estes dois grupos é no caso das interpretadas, como `haskell`, você digita uma expressão escrita corretamente segundo a sintaxe da linguagem, e tem o resultado da expressão *interpretada*, por exemplo

```
ghci> 3 + 4
7
sin 10
-0.5440211108893698
```

quer dizer que você pode usar a linguagem como uma máquina de calcular. No caso das linguagens *compiladas* existe uma pequena diferença que a primeira vista dá a impressão de que elas são mais complicadas. Com elas não é possível obter resultados simples como estes que eu obtive com `haskell` e copiei dentro do texto. É preciso primeiro escrever um programa, usar o compilador da linguagem para produzir um *executável*, e depois *rodar* o executável para obter o resultado final. De fato este segundo processo é um pouco mais longo, mas na prática se termina fazendo a mesma coisa nos dois casos, primeiro se escreve o programa, dentro dum editor de textos, e depois se chama o *compilador* ou o *interpretador* para analisar e rodar o programa.

Mas nas *linguagens interpretadas* é mais amigável o primeiro contacto como você está vendo nos exemplos acima em que fiz uma soma e depois calculei o valor do `seno` de 10.

Então, se você quiser aprender uma linguagem interpretada, como estou aqui lhe apresentado `haskell`, instale o *interpretador*, neste caso é `ghc` e comece a experimentar os comandos, sem medo de errar, ou aprendendo junto com seus próprios erros.

Entretanto, é muito prático produzir um *executável* como um programa compacto que possa ser distribuído pronto para rodar, então com frequência existem compiladores para as *linguagens interpretadas* como é o caso de `haskell` e das outras. O `Java` é outro caso a parte.

Num sistema `Debian/GNU/Linux`, como `Debian`, `Mint` ou o mais conhecido `Ubuntu`, você instala `ghc` com o comando

```
sudo apt-get install ghci
```

2 Primeiros programas em `haskell`

A linguagem `haskell` foge do paradigma comum das linguagens de programação mais conhecidas e usadas que em grande parte se espelham em dois paradigmas que tiveram mais sucesso, `Pascal` e `C`. Alguns comandos de `haskell` são

```
succ 9
[1,2 ..]
fatorial n = if n == 0 then 1 else n*fatorial (n-1)
factorial n = if n == 0 then 1 else product [1 .. n]
product [ ]
sum [ ]
```

1. O primeiro tem como resultado o sucessor de 9, de acordo com os axiomas de Peano,

2. e o segundo produz a lista infinita dos números naturais a partir de 1 e você deve se preparar para acionar `CTRL-C` para não ficar gastando tempo de processamento atoa. E não tem nenhum erro, além disto você vai ter uma ideia do poder da linguagem: ela é capaz de produzir *objetos infinitos* e é por isto que você vai precisar de apertar `CTRL-C` fazendo com que o interpretador pare a construção que nunca iria parar.

3. A terceira expressão define o *fatorial em* `haskell` , recursivamente, e oferece uma ótima oportunidade para mostrar algumas das "esquisitices" da sintaxe de `haskell` se comparadas com outras linguagens de programação.

4. A quarta expressão é uma prova de `haskell` é matemático, estou pedindo que faça o produto dum lista vazia, que por definição é 1. É a mesma razão pela qual o fatorial de zero é 1, o produto duma lista vazia de números inteiros estritamente positivos. A quinta e última é a soma duma lista vazia que também por definição é zero. Compare agora segunda definição que eu fiz de fatorial, e poderia ter sido

   ```
   fatorial n = if n == 0 then product [ ] else product [1 .. n]
   ```

   Se já tiver o `ghci` instalado, copie, cole experimente para ver que o produto duma lista vazia é 1: `fatorial 0`.

- `espaço` é o que separa os símbolos em `haskell`. Em `C`, `Pascal`, `Python`, esta função teria uma definição do tipo `fatorial(n)`, ou pelo menos seria esta a forma de chamar a função depois que ela estivesse definida. Como em `haskell` tudo é função, quando você escreve um símbolo o interpretador espera executar um algoritmo associado àquele simbolo, uma função. Então `fatorial n` vai executar o algoritmo que está associado ao símbolo `fatorial` que precisa que lhe seja passado um número, que deve ser um inteiro positivo do contrário o algoritmo vai se perder. A expressão seguinte está errada:

```
fatorial n = if n == 0 then 1 else n * fatorial n - 1
```

porque depois do espaço "`fatorial n` " terá terminado a definição da função e outra coisa diferente foi passada ao interpretador. Então algumas vezes é preciso usar parêntesis para evitar ambiguidades.

- `if` é uma função da linguagem espera uma expressão que possa ser avaliada como `verdadeira, True` ou `falsa, False`. Por exemplo, você pode experimentar as expressões

```
nM = 3
if nM == 3 then print (nM == 3) else print (nM == 4)
```

Na primeira expressão eu defini "`nM = 3`" ou como é comum dizer nas linguagens de programação "eu dei o valor 3 à variável nM". Possivelmente em `haskell` a maneira correta de falar seja "eu defini nM como 3" por que, como `haskell` é uma linguagem *funcional* o conceito de variável fica desvanecido e "`nM`" é uma função que vale constantemente 3, uma função constante. Claro que posso redefinir esta função como qualquer outra e portanto eu posso redefinir

```
nM = 5
if nM == 3 then print (nM == 3) else print (nM == 4)
```

que você pode executar e analizar o resultado.

3 Algumas funções `haskell`

Será bem pedagógico experimentar as diversas expressões aritméticas ou lógicas que você conhece e analizar o resultado que `ghci` lhe irá oferecer. E não tema errar e nem se assuste com as quatro cinco linhas de reclamação que o interpretador irá lhe oferecer à guisa de repreensão a cada erro. De `enter` que o terminal esquecerá as reprimenda e você poderá voltar a cometer outros erros. E aprenda com os erros.

E se você cometer erros, apenas o interpretador lhe irá dizer isto, mas depois que você der `enter` o sistema continuar disponível para você seguir aprendendo e errando...

Observe uma propriedade bem prática do `ghci`, ele memoriza as expressões que você já tiver digitado, basta usar a seta para cima para fazê-lo repetir as expressões seguidamente. Assim, cometendo algum erro, chame de volta a expressão e a edite corrigindo o erro.

Na tabela seguinte você tem alguns exemplos de funções `haskell` que também aparecem nas demais linguagens de programação para que você se acostume com as diferenças na sintaxe.

succ	+	*	/	-	negate
succ 1	3 + 4	3 * 4	3 / 4	3 - 4	negate 3
'div'	a^b	a + b * c	f(x,y)	sum []	product []
3 'div' 2	2^3	3 + 5 * 2	f x y	sum [1..10]	sum [0.1, 0.3 .. 1]

Execute os exemplos da tabela acima para ver as diferenças da sintaxe do `haskell` com a sintaxe da Matemática ou das linguagens tradicionais de computação. Crie outros exemplos semelhantes.

Embora o uso do terminal do `ghci` seja uma excelente ferramenta para que você se inicialize no estudo da linguagem, melhor mesmo é fazer `haskell` se comunicar com um arquivo porque você

vai programar da mesma forma como se estivesse trabalhando como uma linguagem compilada: escrever programas em arquivos e depois chamar o `haskell` para executá-los. Também você vai reutilizar os seus programas e portanto é preciso que eles estejam arquivados no computador. Eu vou discutir esta etapa no próximo artigo.

Índice Remissivo

Referências

[1] Graham Hutton. *Programming in Haskell*. University of Nottingham, 2004.

[2] David I. Bell Landon Curt Noll and other. Calc - arbitrary precision calculator. Technical report, http://www.isthe.com/chongo/, 2011.

[3] Tarcisio Praciano-Pereira. *Cálculo Numérico Computacional*. Sobral Matematica, 2007.

[4] Thomas Williams, Colin Kelley, and many others. gnuplot, software to make graphics. Technical report, http://www.gnuplot.info, 2010.