

TIPOS DE DADOS E CLASSES INDUTIVAS

RAUL H.C.LOPES

1. INTRODUÇÃO

Você encontra aqui um exemplo típico de tipo de dados recursivo. As seções a seguir apresentam uma definição indutiva para expressões aritméticas e seu equivalente em Haskell. Além disso, apresenta-se um resolvidor de expressões, que usa conceitos de análise léxica e sintática que poderão ser úteis na construção de seu provador de teoremas.

Para extrair o código Haskell deste documento, use o programa **lst2code.pl**, que se encontra no *tarball* a que este documento pertence.

Faça: `lst2code.pl < ae.tex`

Um arquivo de nome **ae.hs** deve ser gerado no diretório corrente. O programa citado extrai todas as seções contidas entre o delimitador de início de código

```
\begin\{lstcode} % file: ae.hs
```

e o delimitador de fim de código

```
\end\{lstcode}
```

concatena-as e grava-as no arquivo de nome **ae.hs**. Note que o que determina o nome do arquivo de saída é o nome **ae.hs**, que se encontra após o texto **%file:**, no delimitador de início de código.¹

2. DEFINIÇÃO INDUTIVA DE EXPRESSÕES

Este documento apresenta uma implementação de um resolvidor de expressões aritméticas. Infelizmente, as expressões representáveis são bastante limitadas: somas de inteiros. Felizmente, o exemplo, deve ser suficientemente ilustrativo do processo de definição de uma linguagem, a das expressões em questão, e de analisadores léxico e sintático para a mesma.

As expressões consideradas podem vistas como pertencentes a uma classe indutiva:

¹Para quem não conhece, **lst2code.pl** está escrito em Perl, uma linguagem de scripting, das mais usadas na Web para implementação de programas que manipulam texto e realizam tarefas administrativas de manutenção de sistemas operacionais e redes de computadores. Bem-venhido à baixaria: se for do interesse da turma, posso dar uma ou duas aulas sobre linguagens de script: perl, awk, bash.

- (i) qualquer inteiro é uma expressão aritmética;
- (ii) a soma de duas expressões aritméticas é uma expressão aritmética;
- (iii) o conjunto de expressões aritméticas é a interseção de todos os conjuntos que se podem gerar usando i e iii.

Expressões só terão vida do ponto-de-vista de programas se:

- tiverem uma representação externa aceitável do ponto-de-vista de seus usuários: seres humanos que se aventurem a usar o programa que deste documento resultará;
- tiverem uma representação interna facilmente digerível pelo interpretador/compilador da linguagem escolhida para implementação: **ghc**, compilador da linguagem **Haskell**.

A linguagem externa das expressões aqui consideradas representará:

- inteiros sem sinal usando base decimal, como seqüências de dígitos: *1*, *12*, *1024*, etc.
- somas sempre delimitados por parênteses: $(1 + 2)$, $((2 + 3) + 5)$, etc.

Refinando a definição anterior, as expressões consideradas podem vistas como pertencentes à classe indutiva:

Definição 1. *(i) se x é inteiro sem sinal, representado na base decimal, então x é uma expressão aritmética;*
(ii) se x e y são expressões aritméticas, então $(x + y)$ é uma expressão aritmética;
(iii) o conjunto de expressões aritméticas é a interseção de todos os conjuntos que se podem gerar usando i e ii.

As funções apresentadas enquadram-se em três classes:

- Funções que determinam se uma expressão dada é formada a partir de elementos básicos válidos do ponto-de-vista da definição 1. Processo conhecido como *análise léxica*.
- Funções que determinam se a seqüência de elementos básicos de uma expressão é sintaticamente válida. Por exemplo, $1 + 2$, embora composta de elementos básicos válidos, os inteiros 1 e 2 e o operador de soma, não é sintaticamente correta por lhe faltar o par de parênteses. Processo conhecido como análise sintática.

Conforme apresentado na figura 1, as expressões, o resolvedor e os analisadores são definidas em um módulo que exporta:

- **Expression**, o tipo de dados de expressões;
- **Token**, o tipo de dados que define os *tokens*, elementos básicos (indivisíveis) das expressões;

```

module InductiveExpr(
    Expression ,
    Token ,
    parse ,
    tokenize ,
    getToken ,
    eval)

```

```

where

```

```

import Char

```

FIGURA 1. A interface do módulo

```

data Token = NoToken
           | TokVal Int
           | TokSum
           | TokOpen
           | TokClose
deriving (Eq,Read,Show)

```

FIGURA 2. O tipo Token

- **tokenize**, função que transforma uma seqüência de caracteres em uma seqüência de tokens;
- **parse**, função que analisa a correção sintática de uma expressão, representada como uma seqüência de *tokens*;
- **eval**, função que avalia uma expressão aritmética.

3. OS ELEMENTOS BÁSICOS VÁLIDOS

O processo de análise léxico da expressão dada é realizado pela função *tokenize*, que converte um *String* em uma seqüência de token, representada como uma lista de tokens: *[token]*.

A representação interna dos tokens considerados válidos é estabelecida pelo tipo *Token*, apresentado na figura 2.

Um tipo de dados é uma representação em sistema formal de alguma entidade abstrata que existe na cabeça do programador que o inventou. Neste a classe *Token* visa enumerar os básicos da linguagem que este escreva imaginou.

- **NoToken** representa a ausência de tokens em uma seqüência vazia;

```

getToken [] = (NoToken, [])
getToken (x:xs) = if isDigit x
                    then getNumber xs (digitToInt x)
                    else if isSumOp x
                        then (TokSum, xs)
                        else if isOpenPar x
                            then (TokOpen, xs)
                            else if isClosePar x
                                then (TokClose, xs)
                                else if isSpace x
                                    then getToken xs
                                    else error ("Token_error_at:" ++ (x:xs))

isSumOp x = x == '+'

isOpenPar x = x == '('

isClosePar x = x == ')'

getNumber [] n = (TokVal n, [])
getNumber (x:xs) n = if isDigit x
                      then getNumber xs ((n*10)+(digitToInt x))
                      else (TokVal n, (x:xs))

```

FIGURA 3. A análise token a token

- **TokVal Int** será usado para construir um token a partir de algum inteiro;
- **TokSum** tem a função representar uma ocorrência do operador de soma;
- **TokOpen** e **TokClose** representam ocorrências de parênteses.

A peça fundamental do processo de análise léxica é implementada pela função **getToken**, veja figura 3, que retira de um string dado o maior prefixo possível que contenha apenas um token, retornando um par composto do token obtido e do sufixo que sobra no string, após a retirado do referido prefixo. Essa função usa função trivias para reconhecimento do operador de soma, de parênteses e de seqüência de ígitos, representando inteiros sem sinal.

Note que **getToken** levanta uma condição de exceção, via função **error**, se algum caracter não admissível aparece no string dado. Além disso, ela considera espaços em branco como characters que podem ser ignorados.

```

tokenize [] = []
tokenize (x:xs) = let
                    (nxttok, rest) = getToken (x:xs)
                    in if nxttok == NoToken
                        then []
                        else nxttok:(tokenize rest)

```

FIGURA 4. O driver da análise léxica

```

data Expression = Val Int | Sum Expression Expression
  deriving (Eq, Read, Show)

```

FIGURA 5. O tipo de dados **Expression**

```

parse xs = let
            (expr, rest) = parseExpr xs
            in if null rest
                then expr
                else error (show rest)

```

FIGURA 6. O driver da análise sintática

A função **tokenize**, figura 4, é o *driver* do processo de análise léxica: dado um string, aplica consecutivamente a função **getToken** até esgotar o string dado para construir um sequência de tokens.

4. A ANÁLISE SINTÁTICA

O processo de análise sintática visa transformar uma sequência de tokens em uma representação interna de expressão aritmética válida, uma expressão aritmética sendo representada internamente através do tipo **Expression**.² **Expression**, veja figura 5, é uma concretização, se é que software é concreto, do conceito abstrato de expressões aritméticas que admitem apenas somas.

O *driver* da análise sintática é a função **parse**, figura 6, que tenta esgotar uma sequência de tokens, usando a função **parseExpr**, transformando-a em uma expressão aritmética única. Não esgotar a sequência de token é um erro: por exemplo

$$(2 + 2)2$$

não é expressão válida, embora $(2 + 2)$ o seja.

²Note como essa representação reflete claramente a definição indutiva de 1, faltando aí apenas o fecho universal, que fica implícito.

```

parseExpr [] = error "a_token_sequence_cannot_represent_an_arithmetic_e
parseExpr ((TokVal x):xs) = (Val x,xs)
parseExpr (TokOpen:xs) = let
                                (ops,rest) = parseClose (parseSum xs)
                                in (opsToSum ops,rest)

parseClose (ops, TokClose:xs) = (ops, xs)
parseClose (ops, xs) = error(show (ops,xs))

opsToSum (op0, op1) = Sum op0 op1

```

FIGURA 7. A análise sintática

A função **parseExpr**, apresentada na figura ??, segue estritamente a forma clássica de definir funções sobre tipos recursivos de dados. Ela mapeia uma seqüência de tokens para **Expression**. Uma seqüência de tokens vazia não é expressão válida. No caso de seqüências não vazias, **parseExpr** retira de uma seqüência de tokens o maior prefixa que representa uma expressão, retorna a representação desse prefixo como **Expression** e o sufixo que resta. A sua definição segue os casos possíveis da definição indutiva 1:

- A expressão pode ser simplesmente um inteiro, caso i.
- A seqüência dada é iniciada por **TokOpen**, indicando a abertura de parênteses que caracteriza o caso ii da definição indutiva. Neste caso, o prefixo esperado na seqüência dada é composto um **TokOpen**, uma seqüência apresentando uma soma e um **TokClose**.

A função **parseSum** recupera os componentes, operandos e operador, de uma soma. Ela é simplesmente a composição da recuperação do primeiro operando, efetuada por **parseSumOp0**, verificação da presença e descarte do operador de soma, efetuada por **parseSumOp1**, e recuperação do segundo operando por **parseSumOp1**. Veja figura ??, que contém código da análise sintática de uma soma.

A função *treeEval* transforma uma **Expression** em um inteiro que representa o valor da expressão associada. Novamente, tem-se aqui um caso de função definida sobre um tipo de dados recursivo.

A avaliação de uma expressão representado em um string é realizada pela função **eval**, que representa somente a composição dos processos **tokenize**, **parse** e **treeEval**. Veja na figura 9 código destas funções.

```

parseSumOp0 :: [Token] -> (Expression, [Token])
parseSumOp1 :: (Expression, [Token]) -> ((Expression, Expression), [Token])
parseSumOptr :: (Expression, [Token]) -> (Expression, [Token])
parseSum = parseSumOp1 . parseSumOptr . parseSumOp0

parseSumOp0 = parseSumOperand

parseSumOp1 (op0, xs) = let
                        (op1, rest) = parseSumOperand xs
                        in ((op0, op1), rest)

parseSumOperand = parseExpr

parseSumOptr (expr, (TokSum:xs)) = (expr, xs)
parseSumOptr xs = error (show xs)

```

FIGURA 8. A análise de somas

```

treeEval (Val x) = x
treeEval (Sum x y) = (treeEval x) + (treeEval y)

eval = treeEval . parse . tokenize

```

FIGURA 9. A função de avaliação

5. ENTÃO?...

As notas acima representam uma tentativa de resumir e ilustrar alguns pontos importantes do curso:

- a relação existente entre o conceito de classe indutiva e tipo abstrato de dados, que passamos a identificar como tipos indutivamente definidos ou tipos recursivos;
- um exemplo de modelagem de tipo de abstrato de dados em uma linguagem de programação;
- a construção de funções recursivas para operar sobre tipos indutivamente definidos;

- o uso de tipos indutivos na modelagem de linguagens, no caso uma linguagem de expressões aritmética simples, e sua análise léxica e sintática.