TUTORIAL HOPE[1]

Roger Bailey, Imperial College, <rb@doc.ic.ac.uk>[2]
Versão estendida do artigo por Sebastian Danicic, Goldsmiths, University of London, <s.danicic@gold.ac.uk>
Tradução e adaptação[3] por Augusto Manzano, IFSP, <augusto.garcia@ifsp.edu.br>

Nota do editor da revista BYTE traduzido e adaptado: Neste artigo, é colocado em negrito a saída dos resultados processados no interpretador Hope para distingui-la da entrada. O interpretador Hope está disponível para download em "hope.manzano.pro.br ou hopelang.blogspot.com para o sistema operacional Windows" ou "github.com/dmbaturin/hope para o sistema operacional Linux".

Nota do tradutor: Algumas partes sofreram adaptações e acréscimos de detalhes ilustrativos não existentes no artigo original para tornar a percepção do texto mais suavizada. Alguns scripts tiveram seus nomes levemente alterados com o uso de apóstrofos após sua identificação para que possam ser executados sem nenhum problema no ambiente Hope.

Em uma linguagem como "*C++*"[4], uma função é uma parte de um programa *embutida* para realizar certa operação padrão, como encontrar raízes quadradas. Para obter a raiz quadrada de um número positivo armazenado em uma variável "x", usa-se a função "sqrt(x)" no trecho do programa onde se deseja obter o valor, como "cout << 1.0 + sqrt(x);". Isso é chamado de *aplicação da função*. O valor representado por "x" é chamado de argumento ou parâmetro real. Desta forma, o programa calcula a raiz quadrada de "x" somando 1.0 ao resultado a ser impresso.

É possível definir as próprias funções especificando como o resultado a ser obtido deve ser calculado a partir de instruções comuns da linguagem em uso. Aqui está, como exemplo, uma função que retorna o maior valor numérico inteiro entre dois valores fornecidos como argumentos:

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

Os identificadores "x" e "y" são chamados parâmetros formais. Eles são usados para nomear os dois valores que são fornecidos como argumentos a função quando esta for chamada. Pode-se usar a função "max" em qualquer lugar em que se precise do retorno de um valor, assim como ocorre com a função "sqrt". Veja como fazer uso da função "max" para filtrar o retorno de valores negativos: "cout << max(z, 0);".

---

1. BAILEY, R. A HOPE TUTORIAL: Using one of the new generation of functional languages. BYTE, New Hampshire, v. 10, n. 8, p. 235-258, ago. 1985.
2. No artigo de Roger Bailey há a indicação de contato junto ao Departamento de Computação do Imperial College. 180 Queens, Gate. Londres SW7 2BZ. Inglaterra.
3. Esta tradução foi adaptada para melhor legibilidade a todos os públicos. Alguns ajustes são definidos para compatibilizar o texto com o idioma português. O texto original está escrito na primeira pessoa do plural. Esta tradução é mantida na segunda pessoa do singular. Vide observação no final deste documento. Outro detalhe importante a contribuição de Sebastian Danicic ao artigo trouxe detalhes a mais, detalhes esses que estão sendo considerados nesta tradução.
4. No artigo o autor menciona a linguagem Pascal. Aqui foi feita uma adaptação para a linguagem C++.

Um caso mais interessante é quando o parâmetro real aplicado a uma função é uma função em si ou envolve a própria função. A função "max" pode ser usada para encontrar o maior valor numérico inteiro entre três números fornecidos, escrevendo "max(a, max(b, c))".

Combinar funções, dessa forma, é chamado de *composição*. Na composição a expressão é avaliada *de dentro para fora* porque a aplicação externa de "max" não pode ser avaliada até que o valor de seu segundo argumento seja conhecido. A aplicação interna de "max" é avaliada primeiro utilizando os valores "b" e "c" e o resultado obtido é usado como parâmetro real para a função "max" mais externa que opera com o valor "a".

Outra forma de combinar funções é definir uma função que seja mais poderosa a partir da definição de outras funções mais simples como sendo *blocos de construção*. Para se obter o maior de três números, basta definir:

```
int MaxOf3(int x, int y, int z)
{
  return max(x, max( y, z ));
}
```

e para seu uso escreva, tão somente "MaxOf3(a, b, c)".

Programação com funções

"*C++*" é uma linguagem de programação imperativa porque os programas escritos nela são receitas de como *fazer algo*. No entanto, se os programas a serem desenvolvidos consistirem apenas no uso de funções, é possível concentrar diretamente sua atenção apenas nos resultados, ignorando por completo como isso é efetivamente calculado. Esqueça que "sqrt" é um trecho de código e pense que "sqrt(x)" é uma forma de escrever um valor em um programa e você terá ideia do que é programação funcional. Você pode pensar assim sobre "MaxOf3" se ignorar a maneira como a função funciona por dentro. A partir da definição de um *kit de ferramentas* com funções úteis e da combinação dessas funções é possível construir programas muito poderosos que são bastante curtos e fáceis de serem entendidos.

Em "*C++*" (sem usar nenhum recurso além do conjunto básico de comandos[5]) as funções só podem retornar objetos de dados *simples*, como números ou caracteres. No entanto, programas reais usam estruturas de dados ao estilo *big data* e não podem ser escritos facilmente usando essas funções. Em Hope as funções podem retornar qualquer tipo de valor incluindo estruturas de dados equivalentes a vetores, matrizes e *structs* de "*C++*" e muito mais. Programar em Hope dá a você o gosto de *escrever a resposta* a partir da definição de uma expressão. Isso proporciona a possibilidade de se definir uma ou mais funções menores como parte da resposta de uma questão maior. Essas funções geralmente não são desenvolvidas como a função interna "sqrt" de "*C++*", pois, para isso, deverão ser escritas pelo próprio programador. Pense nessas funções como definições de objetos de dados e não como algoritmos para serem computados.

Um exemplo simples de Hope - uso de condição

Veja como a função "max" pode ser escrita na linguagem Hope. Como "*C++*", Hope opera com definições de tipos de dados. Isso significa que é necessário informar os tipos de dados de um programa para tornar esses dados consistentes ao seu uso. A definição de uma função deve ser escrita em duas partes. Primeiro, declara-se os tipos dos argumentos e o resultado:

---

[5] Nota do tradutor.

```
dec max : num # num -> num;
```

A cláusula "*dec*" é uma palavra reservada: você não pode usá-la como um nome, diferentemente de "max" que é o nome atribuído a uma função que está sendo definida. Os nomes de funções podem ser definidos a partir do uso de letras maiúsculas e minúsculas (*case-sentive*) e dígitos, devendo o nome de uma função sempre ser iniciado por letra. O estilo mais comum de uso para a definição de nomes para funções são o uso de letras minúsculas. O layout de escrita não é significativo podendo-se separar os símbolos com qualquer número de espaços em branco (ou mesmo sem nenhum espaço em branco, excetuando-se a definição de "dec" e o nome de identificação a frente), tabulações e novas linhas para maior clareza são aceitos. Os símbolos são separados apenas quando podem ser confundidos como se fossem um único nome de identificação, como "dec" e "max", evitando-se algo como "decmax".

A próxima parte da declaração fornece os tipos de argumentos declarados após o símbolo ":" (leia o símbolo ":" como 'para que um'). Os inteiros não negativos são do tipo predefinido "*num*" (em letras minúsculas). O símbolo "*#*" (tralha) deve ser lido como 'e um' (ou fazer uso da palavra reservada "*X*" como substituto de "#"). O símbolo "->" (seta para à direita) deve ser lido como 'retorne'. O ponto-e-vírgula marca o encerramento da declaração na linha de instrução. Toda essa instrução informa ao ambiente de programação que a declaração (dec) da função "max" recebe dois argumentos numéricos (: num # num - para um "num" e um "num"), separados por vírgula os quais retornam um número (-> num;) como resultado.

O resultado de uma função pode ser definido por uma ou mais equações recursivas. A função "max" precisa apenas de uma equação simples para estabelecer sua definição:

```
--- max (x, y) <= if x > y then x else y;
```

Leia os símbolos "---" como 'sendo'. A expressão "max (x, y)" é definida do lado esquerdo da equação a partir dos argumentos "x" e "y" como parâmetros formais ou a partir de nomes de identificação local para atuar sobre os valores fornecidos quando a função for aplicada. Os nomes dos parâmetros são definições locais para a equação, então "x" e "y" não serão confundidos com qualquer outro "x" ou "y" que esteja ativo no programa. O símbolo de asserção "<=" (seta para à esquerda) é lido como 'definida a partir da ação'.

O restante da equação (chamada lado direito) define o resultado da função. Neste exemplo é uma expressão condicional que retorna o maior de dois valores indicados. Os comandos *if*, *then* e *else* são palavras reservadas. A declaração condicional de "*C++*" escolhe entre ações alternativas, mas a expressão condicional em Hope escolhe entre valores alternativos a partir da ótica de que a aplicação de funções são formas de escrever valores em vez de receitas para computá-los. Se o valor da expressão "x > y" for verdadeiro o valor de toda a expressão condicional será o valor "x", caso contrário, será o valor "y".

Quando o valor de uma função é definido por mais de uma expressão, eles são avaliados em uma ordem não especificada. Em um computador adequado, como a máquina ALICE do Imperial College é possível até avaliar ambas as expressões com uma ação de teste em paralelo e descartar um dos valores de acordo com o resultado do teste.

Uso das funções definidas

Um programa Hope é uma expressão única contendo uma ou mais aplicações de função compostas, sendo imediatamente avaliado e fornecendo o resultado de saída, bem como, seu tipo

impresso na tela do terminal. Aqui está um programa simples que usa a função "max" com sua saída indicada na linha seguinte:

```
max(10, 20) + max(1, max(2, 3));
23 : num
```

As regras para avaliar uma expressão são as mesmas usadas na linguagem "*C++*". Os argumentos da função são avaliados primeiro, na sequência as funções são aplicadas e finalmente outras operações são realizadas na ordem de prioridade usual ou estabelecida.

É possível fazer uso de funções existentes para a criação de novas funcionalidades. Aqui está a versão Hope para a função "MaxOf3":

```
dec MaxOf3 : num # num # num -> num;
--- MaxOf3 (x, y, z) <= max (x, max (y, z));
```

Um exemplo mais interessante - uso de repetição

Assim como a declaração condicional em "*C++*" é substituída pelo valor condicional de Hope, a declaração de repetições com laços é substituída pelo valor que está sendo repetido. Aqui está uma função "*C++*" que multiplica, de modo muito simples, dois números inteiros usando adição por repetição iterativa:

```
int mult(int x, int y)
{
  int prod = 0;
  while (y > 0)
  {
    prod = prod + x;
    y = y - 1;
  }
  return prod;
}
```

É difícil ter certeza de que essa função faz adições suficientes (precisei de três tentativas para acertar[6]) e isso parece ser um problema geral com o uso de laços em programas. Uma maneira comum de verificar programas imperativos é simular sua execução. Se for realizado um teste com os valores de entrada de "2" e "3", perceber-se-á que "prod" começa com o valor "0" e obtém valores "2", "4" e "6" em iterações por meio do laço a partir da execução de ações sucessivas, o que sugere que sua definição está correta.

Hope não opera laços, então é necessário escrever todas as ações que o programa "*C++*" executa em uma única expressão. É fácil perceber que isso tem, de certa forma, o número certo de ações necessárias:

```
dec mult : num # num -> num;
--- mult (x, y) <= 0 + x + x + ...
```

ou então saber quantas vezes escrever "+ x". A simulação manual sugere que é preciso escrever "y" um determinado número de vezes (para representar "2 x 3" é necessário que "y" seja escrito "3" vezes[7]), o que torna o teste complicado quando não se conhece o valor exato de “y”. O que

---

[6] Segundo, Roger Bailey.

[7] Nota do tradutor.

se sabe é que para um determinado valor "y" é necessário que as expressões "mult(x, y)" e "mult(x, y - 1) + x" tenham o mesmo número de termos "+ x" se ambas forem escritas por extenso. Desta forma, a segunda expressão possui apenas dois termos, qualquer que seja o valor de "y", então esta será a forma base usada como definição da função "mult":

```
--- mult (x, y) <= mult (x, y - 1) + x;
```

Diante o exposto, vê-se que foi escrito uma solução simplista e até ridícula. Veja que será aplicada a função "mult" para encontrar o próprio valor de "mult". Observe que isso é uma forma abreviada de escrever "0" seguido por "y" e as ocorrências de "+ x". Quando "y" é zero o resultado de "mult" também é zero porque não há termos "+ x". Nesse caso, "mult" não é definida em termos de si mesma, então acrescenta-se na função um teste condicional para determinar facilmente o encerramento. Uma definição aceitável de "mult" é:

```
--- mult (x, y) <= if y = 0 then 0 else mult (x, y - 1) + x;
```

As funções definidas a partir dessa técnica são chamadas de *funções recursivas*. Cada programa "*C++*" que usa laço pode ser escrito com funções recursivas em Hope. Todas as definições recursivas precisam de um caso (chamado de caso base para determinar seu encerramento, também conhecida como aterramento[8]) em que a função não é definida em termos de si mesma como ocorre com os laços em "*C++*" que precisam de uma condição de encerramento.

Outra maneira de usar funções

Hope permite que funções com dois argumentos como "mult" possam ser usadas como um operador infixo. Para isso, é necessário atribuir a função um nível de prioridade e usá-la como um operador aritmético em qualquer situação. A definição de "mult" como operador aritmético infixo é realizada assim:

```
infix mult' : 8;
dec mult' : num # num -> num;
--- x mult' y <= if y = 0 then 0 else x mult' (y - 1) + x;
```

Um número maior na declaração do infixo significa uma prioridade mais alta[9]. O segundo argumento de "mult'" está entre parênteses "(y - 1)" porque sua prioridade "8" é maior do que a operação de subtração embutida. A maioria das funções padrão Hope são operações infixa.

Outros tipos de dados

Hope fornece dois outros tipos de dados primitivos. Um chamado "*truval*" (valor verdade), equivalente ao tipo "bool" de "*C++*" operando a partir dos valores "true" e "false" apresentados junto a expressão "x > y". O operador relacional ">" (maior que) é uma função padrão cujo tipo é "num # num -> truval". Os valores lógicos podem ser usados com expressões condicionais, podendo combiná-los com as funções padrão "*and*", "*or*" e "*not*".

Os caracteres únicos são do tipo "*char*" com valores 'a', 'b' e assim por diante. Os caracteres são úteis como componentes de estruturas de dados na formação de cadeias de caracteres alfanuméricos (*list char*).

---

8 Nota do tradutor.

9 São permitidos o uso de valores entre 1 e 9 (observação do tradutor).

Estruturas de dados

Programas práticos necessitam de estruturas de dados e Hope possui dois tipos padrão já integrados. O tipo mais simples corresponde ao tipo *struct* de "*C++*". É possível vincular um número fixo de objetos de qualquer tipo em uma estrutura chamada tupla. Por exemplo: "(2, 3)" e "('a', true)" são tuplas respectivamente dos tipos "num # num" e "char # truval". Tuplas são usadas em situações quando uma função deve definir o retorno de mais de um valor de uma única vez. Aqui está um exemplo que define a hora do dia considerando o número de segundos desde a meia-noite:

```
dec time24 : num -> num # num # num;
--- time24 (s) <= (s div 3600,
                   s mod 3600 div 60,
                   s mod 3600 mod 60);
```

"*div*" é uma função de divisão de inteiros embutida e "*mod*" fornece o resto após a divisão de inteiros. Se escrito no *prompt* do ambiente de interpretação o uso da função "time24" ocorrerá a apresentação do resultado em uma tupla, além de seu tipo na tela da maneira usual:

```
time24(45756);
(12, 42, 36) : num # num # num
```

O segundo tipo de dados padrão, chamado lista, corresponde aproximadamente a uma matriz unidimensional (vetor) em "*C++*", podendo conter qualquer número de objetos (incluindo nenhum) desde que sejam todos os elementos do mesmo tipo. Desta forma, é possível escrever expressões em programas que representem listas. Por exemplo: "[1, 2, 3]" do tipo "list num".

Há duas funções padrão para a definição de lista, sendo o operador infixo "::" (lido como 'cons' ou 'construtor') que define uma lista em termos de elementos simples em uma lista que contenha o mesmo tipo dos elementos:

```
10 :: [20, 30, 40] de modo que seja [10, 20, 30, 40].
```

Não pense no operador "::" como sendo um recurso que adiciona o elemento "10" à frente da lista "[20, 30, 40]". O que este operador faz é definir uma nova lista "[10, 20, 30, 40]" em termos de dois outros elementos (o elemento "10" e a lita "[20, 30, 40]") sem alterar seu significado da mesma forma que "1 + 3" define um novo valor "4" sem que isso altere o significado de "1" ou "3".

A outra função padrão para a definição de lista é "*nil*" que define uma lista vazia. É possível representar cada lista por uma expressão que consista nas aplicações de "::" e "nil". Quando uma expressão é escrita como

```
['c', 'a', 't']
```

Hope considera que isso é uma forma abreviada de escrever:

```
'c' :: ('a' :: ('t' :: nil))
```

Outro caminho, mais curto, para escrever listas de caracteres é definir uma cadeia de caracteres entre aspas inglesas: "cat".

Quando o resultado de um programa Hope é uma lista, este sempre é impresso entre colchetes. No entanto, sendo uma lista de caracteres esta será impressa entre aspas.

Cada tipo de dado em Hope é definido por um conjunto de funções primitivas como "::" e "nil". Essas funções são chamadas de *funções construtoras* e não são definidas por equações de recursão. Quando uma tupla é definida usa-se indiretamente um construtor padrão chamado "," (leia 'vírgula'). Mais adiante será mostrado como os construtores são definidos para outros tipos de dados.

Funções que definem listas

Para escrever um programa "*C++*" que apresente os primeiros "n" números naturais em ordem decrescente provavelmente seria definido um laço que escreve um valor a cada iteração, como:

```
for (i = n; i >= 1; --i) cout << i;
```

Em Hope escreve-se uma expressão que define todos os valores de uma única vez, como feito na função "mult":

```
dec nats : num -> list (num);
--- nats (n) <= if n = 0 then nil else n :: nats (n - 1);
```

"nil" é um operador útil para escrever o caso base de encerramento de uma função recursiva que define uma lista. Se executada a função no terminal ter-se-á:

```
nats(10);
[10,9,8,7,6,5,4,3,2,1] : list num
```

os números estão em ordem decrescente porque é assim que foram organizamos na lista e não porque foram definidos nessa ordem. Os valores na expressão da lista são tratados como se fossem todos gerados ao mesmo tempo. Na máquina ALICE esses valores são gerados ao mesmo tempo.

Para obter os resultados de um programa Hope na ordem certa (crescente) deve-se colocá-los no lugar certo dentro da estrutura de dados final. Caso haja o interesse em se ter uma lista dos números "n" até "1" em ordem inversa não se pode escrever a definição como:

```
... else n :: nats (n - 1);
```

porque os tipos de argumento para "::" estão ao contrário. É necessário fazer uso de outra abordagem a partir de uma função embutida chamada "<>" (leia 'concatenação') que anexa duas listas em uma. A definição ficará assim:

```
dec nats' : num -> list (num);
--- nats' (n) <= if n = 0 then nil else nats' (n - 1) <> [n];
```

Observe que o valor "n" é colocado entre colchetes ao final da expressão para torná-lo componente de uma lista (lista de um único item) porque o operador de concatenação "<>" exige que ambos os argumentos sejam listas. Também pode-se escrever "(n :: nil)" em vez de " [n]".

Estruturas de dados como parâmetros

Suponha a existência de uma lista de números inteiros para a qual se deseja escrever uma função que some todos os seus elementos. A declaração será semelhante a esta:

```
dec sumlist : list (num) -> num;
```

Atente para o fato de que é necessário referir-se aos elementos individuais do parâmetro real nas equações que definem a lista de soma, isso usando uma equação cujo lado esquerdo se parece com isto:

```
--- sumlist (x :: y) ...
```

Esta é uma expressão que envolve construtores de lista que correspondem a um parâmetro real que é a própria lista. Os componentes "x" e "y" são parâmetros formais, agora usados para nomear partes individuais do valor do parâmetro real. Em uma aplicação da função "sumlist" como:

```
sumlist([1,2,3])
```

o parâmetro real é desmembrado de forma que "x" é associado ao número "1" e "y" é associado ao restante da lista como "[2, 3]". A equação completa fica definida:

```
--- sumlist (x :: y) <= x + sumlist (y);
```

Observe que não há a definição de um teste de caso base para o encerramento da ação. Neste caso, a verificação de lista vazia é a condição definida. No entanto, não é possível fazer seu teste diretamente nessa equação porque não há nenhum parâmetro formal que faça referência a lista toda. Na verdade, se for definida a chamada:

```
Sumlist (nil);
```

será retornado uma mensagem de erro porque não é possível fazer com que "nil" encontre os valores de "x" e "y". Neste aspecto, é importante cobrir este caso separadamente usando uma segunda equação de recursão:

```
--- sumlist (nil) <= 0;
```

As duas equações, em tese, podem ser fornecidas em qualquer ordem. Quando na ação da função "sumlist" é aplicado o parâmetro real, este é examinado para ver qual construtor da função foi usada para sua definição. Se o parâmetro real for uma lista não vazia a ação a ser executada será "x + sumlist (y)" porque as listas não vazias são definidas usando o construtor "::". O primeiro número da lista é denominado "x" (cabeça da lista) e o restante da lista "y" (cauda da lista). Se o parâmetro real for uma lista vazia, a ação executada será "sumlist (nil) <= 0" porque as listas vazias são definidas usando o construtor "nil".

OBS: *É importante fazer uma ressalva ao parágrafo anterior. O trecho sinalizado "em tese" foi acrescido pelo tradutor para chamar a atenção a uma questão. Na época de publicação deste tutorial seu autor não levou em consideração uma regra que é utilizada nos dias atuais a publicação desta tradução que ocorreu no primeiro semestre de 2021. Dentro das regras de aplicação de operações recursivas o caso base deve ser definido antes da operação principal da função e não após. Mesmo que uma linguagem funcional aceite a colocação das equações em qualquer posição da função a escrita realizada não fará isso: http://www.facom.ufu.br/~madriana/PF/recursao.pdf. Veja o código por completo:*

```
dec sumlist : list (num) -> num;
--- sumlist (nil) <= 0;
--- sumlist (x :: y) <= x + sumlist (y);
```

Correspondência de padrões

A expressão composta de construtores que aparece ao lado esquerdo de uma equação de recursão é chamada de *padrão*. Selecionar a equação de recursão correta e desmontar o parâmetro real para nomear suas partes é chamado de *correspondência de padrões*. Ao ser escrita uma função é necessário fornecer uma equação de recursão para cada construtor possível definindo-se seu tipo de argumento.

Às vezes, não é preciso desmontar o parâmetro real, basta usar um parâmetro formal no padrão que corresponda a todo o objeto, independentemente de quais construtores foram usados para defini-lo. Como exemplo, veja a definição de uma função que efetue a concatenação de duas listas como ocorre com o uso do operador "<>":

```
infix cat : 4;
dec cat : list (num) # list (num) -> list (num);
--- nil cat r <= r;
--- (h :: t) cat r <= h :: (t cat r);
```

A primeira equação de recursão cobre o caso em que a primeira lista é vazia. O segundo parâmetro de lista é correspondido pelo padrão "l" esteja vazio ou não. O primeiro parâmetro da segunda equação da lista é correspondido a partir de "(h :: t) " de forma que seu primeiro elemento (a cabeça, argumento "h") e a lista restante (a cauda, argumento "t") podem ser referenciados separadamente no lado direito da equação.

Além de escrever equações de recursão suficientes para satisfazer todos os construtores de parâmetros é importante ter cuidado para não escrever conjuntos de equações onde mais de um padrão pode corresponder aos parâmetros reais porque isso criaria ambiguidade.

É possível escrever padrões que combinem com argumentos que sejam tuplas da mesma forma usando o construtor de tupla ",". Quando escrita a função "mult (x, y)" você provavelmente pensou que os parênteses e a vírgula tinham algo a ver com a aplicação da função. Na verdade, o que ocorreu foi a construção de uma tupla e os parênteses só foram necessários porque os argumentos possuem baixa prioridade. Hope trata todas as funções como tendo apenas um argumento podendo ser uma tupla quando se deseja o efeito de vários argumentos. Sem parênteses

```
mult x, y
```

seria interpretado como

```
(mult (x), y)
```

O pedido, por exemplo, de "(mult2 (2), 3)" devolve como resposta um erro de tipo[10]

[10] Trecho acrescido pelo tradutor para melhor ilustração.

```
  mult 2
  mult : num # num -> num
  2 : num
type error - argument has wrong type
```

informando que o erro de tipo refere-se ao uso de algum argumento com o tipo errado "type error - argument has wrong type". Além da mensagem de erro é informado que a função deve receber uma tupla de dois argumentos "num # num" para devolver uma resposta "-> num" coerente a entrada, indicando que o erro ocorre no argumento "2" em "mult 2" que retorna como saída "2 : num" quando deveria ser um número "6" a partir da entrada "(2, 3)".

Uma equação de recursão com o lado esquerdo:

```
--- mult (x, y) <= ...
```

é apenas uma correspondência de padrão em uma tupla. O primeiro item na tupla é denominado "x" e o segundo "y".

A correspondência de padrões também pode ser usada com parâmetros "num", definidos por dois construtores chamados "succ" e "0". O construtor "succ" define um número em termos do próximo mais baixo e o construtor "0" não tem argumentos e define o valor zero. Certamente "0" é um valor e não uma função? Bem, aplicações de função são outra forma de escrever valores, então, é bastante consistente pensar em "0" como uma aplicação de função. Veja agora uma versão de “mult" que usa correspondência de padrões para identificar o caso base:

```
infix mult'' : 8;
dec mult'' : num # num -> num;
--- x mult'' 0       <= 0;
--- x mult'' succ (y) <= (x mult'' y) + x;
```

Considere "succ (y)" como o sucessor de algum número "y"'. Em vez de nomear o parâmetro "y" real como feito na versão original de "mult" faz-se aqui a nomeação de seu predecessor.

Expressões simplificadoras

Em programas "*C++*" é possível simplificar expressões complexas removendo subexpressões comuns e avaliando-as separadamente. Ao invés de "cout << (x + y) * (x + y);" provavelmente seria escrito "z = x + y; " e "cout << z * z;" o que é mais claro e eficiente. Os programas Hope consistem apenas em expressões que podem ser simplificadas. Isto é feito com o uso de *expressão qualificada*:

```
let z == x + y in z * z;
```

Parece a definição de uma igualdade, mas não é. O operador "==" deve ser lido como 'definido como' e “z” é local para o armazenamento temporário da ação da expressão após o comando “*in*”. Se for escrito algo como:

```
let z == z + 1 in z * z;
```

na verdade o que ocorre é a definição de uma nova variável "z" a partir de "*let*" para ser usada na subexpressão "z * z" ocultado a subexpressão "z + 1".

Há uma segunda forma de expressão qualificada destinada a pessoas que gostam de usar variáveis primeiro e definir seus significados depois:

```
z * z where z == x + y;
```

O resultado da expressão qualificada é o mesmo, seja utilizando-se "let" ou "*where*". A operação "x + y" é avaliado primeiro e seu valor é usado na expressão principal.

A expressão de qualificação geralmente será uma aplicação de função que define uma estrutura de dados. Desejando nomear parte da estrutura pode-se usar um padrão no lado esquerdo do símbolo "==":

```
dec time12 : num -> num # num ;
--- time12 (s) <= (if h > 12 then h-12 else h, m) where
                     (h, m, s) == time24 (s);
```

Essa forma de construção é usada com maior frequência quando funções recursivas são escritas para definir tuplas. Aqui está um exemplo típico. Suponha que haja a necessidade de formar uma sequência de palavras a partir de uma frase. Para simplificar, uma palavra é considerada qualquer sequência de caracteres. As palavras são separadas na frase por qualquer número de espaços em branco. A frase e uma única palavra serão do tipo “list (char)” e a sequência final de palavras uma “list (list char)”.

É bastante simples obter a primeira palavra. Esta é uma função que faz isso:

```
dec firsttry : list (char) -> list (char);
--- firsttry (nil)    <= nil;
--- firsttry (c :: s) <= if c = ' '
                           then nil
                           else c :: firsttry (s);
```

Um dos recursos interessantes de Hope é a capacidade de poder receber e apresentar qualquer tipo de valor> Assim sendo, é fácil verificar funções individualmente. Para ver o funcionamento da função execute

```
firsttry("You may hunt it with forks and Hope");
"You" : list char
```

Mas há um problema aqui. Observe que é necessário operar o restante da frase caso queira encontrar as demais palavras. Nesse sentido, é necessário providenciar para que a função retorne a lista restante, bem como a primeira palavra. É aqui que entram as tuplas:

```
dec firstword : list (char) -> list (char) # list (char);
--- firstword (nil)    <= (nil, nil);
--- firstword (c :: s) <= if c = ' '
                            then (nil, s)
                            else ((c :: w, r) where
                                (w, r) == firstword (s));
```

A expressão qualificada está entre parênteses, portanto, só se aplica à expressão após o "else", caso contrário, será avaliada a primeira palavra recursivamente desde que a frase não esteja vazia mesmo que comece com um espaço em branco. Esta versão da função produz:

```
firstword("Hope springs eternal ...");
```

```
("Hope", "springs eternal ...") : list char # list char
```

A partir desta ocorrência pode-se definir uma função para dividir a frase em uma lista de palavras individuais:

```
dec wordlist : list (char) -> list (list (char));
--- wordlist (nil)    <= nil;
--- wordlist (c :: s) <= if c = ' '
                           then wordlist (s)
                           else (w :: wordlist (r) where
                               (w, r) == firstword (c :: s));
```

que pode ser testado no terminal solicitando:

```
wordlist("While there's life there's Hope");
["While", "there's", "life", "there's", "Hope"] : list (list char)
```

Até agora os recursos apresentados de Hope foram discutidos a partir de algo em comum com linguagens tradicionais, como "*C++*", mas sem muitas de suas limitações como as estruturas de dados de tamanho fixo. Até este ponto você tem a noção do uso do estilo funcional de programação em que os programas não são mais receitas de ação, mas sim, definições de objetos sobre os dados.

Na sequência são apresentados recursos de Hope que levam esta apresentação a um nível mais alto, permitindo escrever programas que não só são extremamente poderosos e concisos, mas que podem ser verificados quanto à exatidão em tempo de execução e mecanicamente menores, sendo esses programas transformados em versões mais eficientes.

Apresentação de funções polimórficas

O ambiente Hope pode detectar muitos tipos comuns de erros verificando os tipos de todos os objetos nas expressões. Isso é mais difícil do que realizar essas verificações em tempo de execução, mas é mais eficiente e evita o constrangimento de descobrir um erro em tempo de execução em uma ramificação raramente executada do sistema, por exemplo, de controle de tráfego aéreo que acabou de ser escrito.

No entanto, a verificação de tipo estrita pode ser incômoda se houver a necessidade de realizar alguma operação que não dependa do tipo dos dados. Tente escrever uma sub-rotina em "*C++*" para reverter um *array* de 10 inteiros ou 10 caracteres e você verá o que quero dizer.

Hope evita esse tipo de restrição permitindo que uma função opere em mais de um tipo de objeto. Já foram usados os construtores padrão "::" e "nil" para definir uma "list (num)", uma "list (char)" e uma "list (list char)". A função de igualdade padrão "=" comparará quaisquer dois objetos do mesmo tipo. As funções com essa propriedade são chamadas polimórficas. As funções internas de "*C++*" como "abs" e "sqrt" e também as operações ">" e "=" são polimórficas de uma forma primitiva.

Em Hope é possível definir as próprias funções polimórficas. A função "cat" anterior concatena apenas listas de números, mas poderia ser usada para listas contendo qualquer tipo de objeto. Para fazer isso, primeiro é necessário definir um *tipo universal* de dado chamado de *variável de tipo*. Isso poderia ser usado na declaração de “cat” como uma função que aceite qualquer tipo de dado.

```
typevar alpha;
infix cat' : 8;
dec cat' : list (alpha) # list (alpha) -> list (alpha);
```

Isso diz que "cat" tem dois parâmetros que são listas e define uma lista, mas não diz que tipo de objeto está nessa lista. No entanto, *"alfa"* sempre representa o mesmo tipo em uma determinada declaração, portanto, todas as listas devem conter o mesmo tipo de objeto. As expressões:

`[1,2,3] cat' [4,5,6]` e `"123" cat' "456"`

são aplicações válidas da função "cat'" permitindo definir uma "list (num)" e uma "list (char)" respectivamente, enquanto a expressão:

```
[1,2,3] cat' "456"
```

não é, uma vez que alfa não pode ser interpretado como dois tipos diferentes. A interpretação de uma variável de tipo é local para uma declaração, portanto, ela pode ter diferentes interpretações em outras declarações sem que ajam confusão[11].

É claro que só faz sentido que uma função seja polimórfica desde que as equações que a definem não façam suposições sobre os tipos. No caso de "cat'" a definição usa apenas "::" e "nil" que são polimórficos. No entanto, uma função como "sumlist" usa o operador "+" que denota que só pode ser usada com listas de números como parâmetros.

Definição dos próprios tipos de dados

Tuplas e listas são muito poderosas, mas para funções mais sofisticadas é necessário definir os próprios tipos. Os tipos definidos pelo usuário tornam os programas mais claros e ajudam o verificador de tipos a auxiliar o programador. Observe a definição de um novo tipo de dados em uma declaração de dados:

```
data vague == yes ++ no ++ maybe;
```

O comando "*data*" é uma palavra reservada e "vague" é o nome do novo tipo. O operador "==" neste contexto é lido como 'é definido como' e o operador "++" é lido como 'ou'. Os rótulos "yes", "no" e "maybe" são os nomes das funções construtoras para o novo tipo. Agora é possível escrever definições de funções que usam esses construtores em padrões:

```
dec evade : vague -> vague;
--- evade (yes)   <= maybe;
--- evade (maybe) <= no;
```

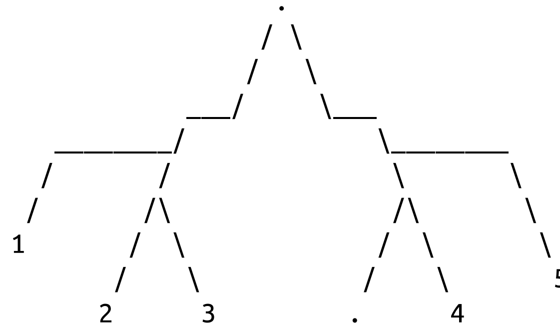
Os construtores podem ser parametrizados com qualquer tipo de objeto, incluindo o tipo que está sendo definido. É possível definir tipos como listas cujos objetos são de tamanho ilimitado usando no tipo ações de definição recursiva. Como exemplo, aqui está uma árvore binária definida pelo usuário que pode conter números como suas folhas:

---

[11] No artigo publicado na revista BYTE há uma nota do editor informando que a versão do interpretador Hope disponibilizada para os testes dos scripts no serviço *BYTEnet* (plataforma para downloads da revista BYTE) possuía de forma predefinida os tipos "alpha" e "beta". O mesmo pode ser dito a respeito dos interpretadores disponibilizados para os sistemas operacionais Windows e Linux que são baseados no código fonte do interpretador escrito pelo Professor Ross Paterson da City, University of London.

```
data tree == empty ++ tip (num) ++ node (tree # tree);
```

Existem na definição "tree" três construtores: "empty" que define uma árvore sem nada nela; "tip" define uma árvore em termos de um único "num" e "node" define uma árvore em termos de duas outras árvores. Aqui está uma árvore típica:



Aqui está um exemplo de uma função que manipula árvores. Ele retorna a soma de todos os números contidos na estrutura:

```
dec sumtree : tree -> num;
--- sumtree (empty)       <= 0;
--- sumtree (tip (n))     <= n;
--- sumtree (node (l, r)) <= sumtree (l) + sumtree (r);
```

Infelizmente não há um atalho para escrever constantes de árvore como há para constantes de lista. Só resta escrevê-las por um longo caminho usando construtores. Para fazer uso da função "sumtree" para somar todos os números na árvore do exemplo acima deve-se escrever a expressão:

```
sumtree(node(node(tip(1),
                  node(tip(2),
                       tip(3))),
             node(node(empty,
                       tip(4)),
                  tip(5))));
```

Isso não é realmente uma desvantagem porque programas que manipulam estruturas de dados complexas como árvores geralmente as definem usando outras funções. No entanto, é muito útil poder informar qualquer tipo de estrutura de dados constante no terminal quando se está verificando uma função individual como "sumtree". Para testar um programa "*C++*"aos poucos geralmente torna-se necessário escrever elaborados arranjos de teste ou definir trechos específicos para gerar dados de teste.

O identificador "list" não é realmente um tipo de dado Hope. Este identificador é chamado de 'construtor de tipo' e deve ser parametrizado com um tipo real antes de representar um dado. Isto é feito em todas as vezes que se declara uma "list (num)" ou uma "list (char)". O parâmetro também pode ser um tipo definido pelo usuário como em uma "list (tree)" ou mesmo uma variável de tipo como em "list (alfa)" que define um tipo de dado polimórfico. A propósito, construir novos tipos de dados como esse é uma operação de tempo de compilação e não deve ser confundida com a construção de novos valores de dados que são uma operação de tempo de execução.

Você pode definir seus próprios tipos de dados polimórficos. Esta é uma versão da árvore binária que definida anteriormente que pode ter qualquer tipo de valor em suas folhas:

```
data tree' (alpha) == empty' ++
                      tip' (alpha) ++
                      node' (tree' (alpha) # tree' (alpha));
```

Mais uma vez "alfa" é considerado o mesmo tipo em uma instância de uma árvore. Se for um número, todas as referências à árvore (alfa) são consideradas referências à árvore (num).

É possível definir funções polimórficas que operam em árvores contendo qualquer tipo de objeto porque os construtores de árvore agora são polimórficos. Aqui está uma função para "achatar" uma árvore binária em uma lista do mesmo tipo de objeto:

```
dec flatten : tree' (alpha) -> list (alpha);
--- flatten (empty')      <= nil;
--- flatten (tip' (x))    <= x :: nil;
--- flatten (node' (x,y)) <= flatten (x) <> flatten (y);
```

A função "flatten" pode achatar diversos tipos de árvore:

```
flatten(node(tip(1), node(tip(2), tip(3))));
[1,2,3] : list num

flatten(node(tip("one"),
            node(tip("two"),
                  tip("three"))));
["one","two","three"] : list (list char)

flatten(node(tip(tip('a')),
            node(tip(empty),
                  tip(node(tip('c'),
                              empty)))));
[ tip'a',empty,node(tip'c',empty) ] : list (tree char)
```

Observe como o verificador de tipo pode necessitar passar por vários níveis antes de deduzir o tipo do resultado.

Programas mais concisos

A importância dos tipos e funções polimórficas é que nos permitem escrever programas mais curtos e claros. É semelhante à maneira como as sub-rotinas em "*C++*" ao permitir usar o mesmo código para operar em diferentes valores de dados, mas mais poderosos. Desta forma, pode-se escrever uma única função Hope para reverter uma lista de números ou caracteres, onde seria necessário escrever duas sub-rotinas "*C++*" idênticas para reverter um array de inteiros e um array de caracteres, caso não sejam usadas definições de macros[12].

As funções polimórficas podem sempre ser usadas quando há a preocupação voltada a organização dos objetos em uma estrutura de dados e não com seus valores. Às vezes, é necessário aplicar alguma função aos itens de dados primitivos na estrutura. Aqui está uma função que usa

[12] Observação do tradutor.

um segundo quadrado de função para definir uma "list (num)" cujos elementos são os quadrados de outra "list (num)":

```
dec square : num -> num;
--- square (n) <= n * n;

dec squarelist : list (num) -> list (num);
--- squarelist (nil)    <= nil;
--- squarelist (n :: l) <= square (n) :: squarelist (l);
```

Cada vez que for escrita uma função para processar cada elemento de uma lista, estará sendo escrito algo quase idêntico a "squarelist". Esta é uma função para definir uma lista de fatoriais:

```
dec fact : num -> num;
--- fact (0)        <= 1;
--- fact (succ (n)) <= succ (n) * fact (n);

dec factlist : list (num) -> list (num);
--- factlist (nil)    <= nil;
--- factlist (n :: l) <= fact (n) :: factlist (l) ;
```

A função "factlist" tem exatamente a mesma 'forma' que a função "squarelist". Possuem como diferença apenas a aplicação "fact" ao invés de "square" recursivamente. Os valores que diferem entre os aplicativos geralmente são fornecidos como parâmetros reais. Hope trata as funções como objetos de dados, então pode-se fazer isso de uma forma perfeitamente natural. Uma função que pode assumir outra função como um parâmetro real é chamada de *função de ordem superior*. Quando feita sua declaração deve-se fornecer o tipo de parâmetro formal que representa a função da maneira usual. A declaração de fato nos diz que é "num -> num". Leia isso como *uma função que mapeia números em números*.

Agora veja como essa ideia pode ser usada para escrever lista de fatoriais e lista de quadrados com uma única função de ordem superior. A nova função precisa de dois parâmetros, a lista original e a função aplicada sobre a lista. Sua declaração será

```
dec alllist : list (num) # (num -> num) -> list (num);
```

O formato da função "alllist" é o mesmo usados nas funções "factlist" e "squarelist", mas a função aplicada a cada elemento da lista será o parâmetro formal:

```
--- alllist (nil, f)    <= nil;
--- alllist (n :: l, f) <= f (n) :: alllist (l, f);
```

A função "alllist" deve ser usada a partir da ação:

```
alllist([2,4,6], square);
[4,16,36] : list num

alllist([2,4,6], fact);
[2,24,720] : list num
```

Observe que não há lista de argumentos após "square" ou "fact" na aplicação de "alllist". Veja que esta construção não será confundida com composição funcional em uso. A função "fact(3)" representa uma aplicação da função, mas o uso de "fact" por si só representa a função não avaliada.

As funções de ordem superior também podem ser polimórficas. Essa ideia pode ser usada para escrever uma versão mais poderosa de "alllist" que aplicará uma função arbitrária a cada elemento de uma lista de objetos de tipo arbitrário. Essa versão da função geralmente é conhecida como "map":

```
typevar alpha, beta;
dec map : list (alpha) # (alpha -> beta) -> list (beta);
--- map (nil, f)    <= nil;
--- map (n :: l, f) <= f (n) :: map (l, f);
```

A definição agora usa duas variáveis de tipo "alfa" e "beta". Cada um representa o mesmo tipo real em uma instância do "map" mas os dois tipos podem ser diferentes. Isso significa que pode-se usar qualquer função que mapeie *dados alfa* em *dados beta* para gerar uma lista de *betas* a partir de qualquer lista de *alfas*.

Os tipos reais não estão restritos a escalares o que torna o mapa muito mais poderoso do que se pode perceber à primeira vista. Suponha a existência de uma função polimórfica que encontre o tamanho de uma lista (a quantidade de elementos existentes):

```
typevar gamma;
dec length : list (gamma) -> num;
--- length (nil)    <= 0 ;
--- length (n :: l) <= 1 + length (l);

length ([2,4,6,8]) + length("cat");
7 : num
```

A função "map" pode ser usada para aplicar o comprimento de cada elemento de uma lista de palavras definidas pela lista de palavras:

```
map(wordlist("The form remains, the function never dies"), length);
[3, 4, 8, 3, 8, 5, 4] : list num
```

Neste exemplo "alfa" é considerado uma "list (char) " e "beta" um "num", então o tipo da função deve ser "(list (char) -> num)". O comprimento é adequado se "gama" for considerada um caractere.

Padrões comuns de recursão

A função "map" é poderosa porque resume um padrão de recursão que aparece com frequência nos programas Hope. Veja outro padrão comum na função "length" usada anteriormente. Aqui está outro exemplo do mesmo padrão:

```
dec sum : list (num) -> num;
--- sum(nil)    <= 0 ;
--- sum(n :: l) <= n + sum (l);
```

O padrão subjacente consiste em processar cada elemento da lista e acumular um único valor que forma o resultado. Em suma, cada elemento contribui com seu valor para o resultado final. Na função "length" a contribuição é sempre "1" independentemente do tipo ou valor do elemento. No entanto, o padrão é idêntico. As funções que exibem esse padrão de comportamento são do tipo "(list (alpha) -> beta)".

Na definição da função a equação para um parâmetro de lista não vazia especificará uma operação cujo resultado é um "beta" quando em uso o operador "+" no caso das funções "length" e "sum". Um argumento da operação será um elemento de lista e o outro será definido por uma chamada recursiva, portanto, o tipo da operação deve ser:

```
(alpha # beta -> beta)
```

Esta operação difere entre suas aplicações, portanto, será fornecida como um parâmetro. Finalmente é necessário ter um parâmetro do tipo "beta" para especificar o resultado do caso base. A versão final da função geralmente é conhecida como "reduce". Na definição a seguir o símbolo "!" introduz um comentário que termina com outro símbolo (não obrigatório) "!" ou com uma nova linha:

```
dec reduce : list (alpha) #                ! lista de entrada         !
                  (alpha # beta -> beta) # ! a operação de redução    !
                  beta                     ! o resultado do caso base !
                  -> beta;                 ! o tipo de resultado      !
--- reduce (nil, f, b)    <= b ;
--- reduce (n :: l, f, b) <=  f(n, reduce (l, f, b));
```

Para usar a função "reduce" como substituição para a função "sum" é necessário fornecer a função padrão o operador "+" como parâmetro real. Isso pode ser feito, desde que a operação seja prefixada com comando "*nonop*" para dizer que não se quer usá-lo como um operador infixo mas, simplesmente, como uma função:

```
reduce([1,2,3], nonop +, 0);
6 : num
```

Quando se utiliza a função "reduce" como substituição para "length" não se tem interesse no primeiro argumento da operação de redução porque é sempre adicionado "1", qualquer que seja o elemento da lista. Esta função ignora seu primeiro argumento:

```
dec addone : alpha # num -> num;
--- addone (_, n) <= n + 1;
```

O uso do argumento coringa "_" (símbolo *underscore*) permite representar qualquer argumento ao qual não se quer fazer referência valendo para tanto qualquer valor que seja atribuído.

```
reduce("a map they could all understand", addone, 0);
31 : num
```

Como a função "map" a função "reduce" é muito mais poderosa do que parece à primeira vista. A função de redução não precisa definir um escalar. Aqui está um candidato que insere um elemento em uma lista ordenada do mesmo tipo do elemento:

```
dec insert : alpha # list (alpha) -> list (alpha);
--- insert (i, nil)    <= i :: nil;
--- insert (i, h :: t) <= if i < h
                            then i :: (h :: t)
                            else h :: insert (i, t);
```

Na verdade isso não é estritamente polimórfico como sua declaração sugere, porque usa a função embutida "<" que só é definida sobre números e caracteres, mas mostra o tipo de coisa que se pode fazer quando se reduz uma lista de caracteres:

```
reduce("All sorts and conditions of men", insert, nil);
"     Aacddefiillmnnnnoooorssstt" : list char
```

veja que isso realmente os classifica. O método de classificação (classificação por inserção) não é muito eficiente, mas o exemplo mostra algo do poder das funções de ordem superior, em particular, sobre a função "reduce". É até possível usar a função "reduce" para obter o efeito de ação da função "map" mas isso é fica como um exercício para você leitor, como dizem.

É claro que as funções "map" e "reduce" trabalham apenas com "list (alfa)". Será necessário fornecer versões diferentes para os tipos estruturados customizados pelo usuário. Este é o estilo preferido na programação Hope, porque torna os programas amplamente independentes em relação a "forma" das estruturas de dados que eles usam. Aqui está um tipo alternativo de árvore binária que contém dados em seus nós em vez de estarem esses dado suas pontas:

```
data tree'' (alpha) == empty'' ++
                       node'' (tree'' (alpha) # alpha #
                       tree'' (alpha));

dec redtree : tree'' (alpha) #
              (alpha # beta -> beta) #
              beta -> beta;
--- redtree (empty'', f, b)          <= b;
--- redtree (node'' (l, v, r), f, b) <=
              redtree (l, f, f (v, redtree (r, f, b)));
```

Esse tipo de arvore pode ser utilizada para definir uma operação mais eficiente. Uma árvore binária ordenada tem a propriedade de que todos os objetos em sua subárvore esquerda precedem logicamente o objeto de nó e todos aqueles em sua subárvore direita são iguais ao objeto de nó ou o sucedem logicamente. É possível construir uma árvore ordenada se houver uma função que efetua a inserção de novos objetos em uma árvore já ordenada, como:

```
dec instree : alpha # tree'' (alpha) -> tree'' (alpha);
--- instree (i, empty'')          <= node'' (empty'', i, empty'');
--- instree (i, node'' (l, v, r)) <=
              if i < v
               then node'' (instree (i, l), v, r)
               else node'' (l, v, instree (i, r));
```

Uma lista pode ser classificada a partir da conversão de seus elementos em uma árvore ordenada usando "instree" e, em seguida, achatando a árvore de volta em uma lista. Isso é muito fácil de especificar usando os dois tipos de redução já definidas:

```
dec sort : list (alpha) -> list (alpha);
--- sort (l) <= redtree (reduce (l, instree, empty''), nonop ::, nil);

sort("Mad dogs and Englishmen");
"   EMaadddegghilmnnnoss" : list char
```

Funções anônimas

Quando foram usadas as funções "map" e "reduce" foi necessário definir funções extras como "fact" e "square" para serem passadas como parâmetros. Isso é um incômodo caso essas fun-

ções não sejam necessárias em algum outro lugar do programa, especialmente, se forem funções triviais, como "sum" ou "addone". Para uso imediato em casos como este pode-se lançar mão de função anônima chamada de *expressão lambda*. Aqui está uma expressão lambda correspondente à soma:

```
lambda (x, y) => x + y
```

O símbolo *lambda* apresenta a função com "x" e "y" como seus parâmetros formais. A expressão "x + y" é a definição do resultado retornado da função. A parte após lambda é apenas uma equação de recursão indicada com "=>" em vez de "<=". Aqui está outra expressão lambda usada como o parâmetro real da função "reduce":

```
reduce(["toe","tac","tic"], lambda (a, b) => b <> a, nil);
"tictactoe" : list char
```

Pode haver mais de uma equação de recursão na expressão lambda, desde que sejam separadas uma da outra pelo símbolo "|", sendo a correspondência de padrões usada para selecionar a ação apropriada. Aqui está um exemplo que usa correspondência de padrões em uma expressão lambda para evitar a divisão por zero quando a função que ela define é executada:

```
map([1,0,2,0,3], lambda(0) => 0 | (succ(n)) => 100 div succ(n));
[100,0,50,0,33] : list num
```

Funções que criam novas funções

Como visto as funções Hope possuem *direitos totais* e podem ser passadas como parâmetros como qualquer objeto de dados. Não será nenhuma surpresa saber que é possível retornar uma função como resultado de outra função. O resultado pode ser uma função nomeada ou uma função anônima a partir de uma expressão lambda. Aqui está um exemplo simples:

```
dec makestep : num -> (num -> num);
--- makestep (i) <= lambda (x) => i + x;

makestep(3);
lambda x => 3 + x : num -> num
```

Como percebido ao fazer uso da função "makestep" seu resultado é uma função anônima que adiciona uma quantidade fixa a seu único argumento. O tamanho do incremento foi especificado como um parâmetro real a ser executado quando a nova função foi criada e tornou-se *vinculada* à sua definição. Se for usada a instrução a seguir ver-se-á que ela realmente adiciona "3" ao seu parâmetro real:

```
makestep (3) (10);
13: num
```

Existem duas aplicações aqui. Primeiro é aplicado "makestep" a "3" e, em seguida, a função anônima resultante é aplicada a "10". Finalmente, aqui está uma função que tem funções tanto como parâmetro real quanto como resultado:

```
dec twice : (alpha -> alpha) -> (alpha -> alpha);
--- twice (f) <= lambda (x) => f (f (x));
```

A função "twice" define uma nova função que tem um único argumento e alguma outra função "f" vinculada à sua definição. A nova função tem o mesmo tipo que "f". É possível ver seu efeito usando uma função simples como "square":

```
twice(square);
lambda x => square (square x) : num -> num

twice(square) (3);
81 : num
```

Conclusão

Neste artigo foi lhe apresentado às ideias de programação funcional por meio de uma das novas gerações de linguagens funcionais. Você viu como um programa Hope é apenas uma série de funções que são consideradas como definições de partes de uma estrutura de dados – *os resultados* do programa - e como a poderosa ideia de funções de ordem superior permite capturar muitos padrões de programa em uma única função.

Algumas dessas ideias são familiares aos usuários da linguagem LISP, mas aparecem de forma mais pura no Hope, porque não existem mecanismos para atualizar as estruturas de dados como SETQ e RPLACA ou para especificar a ordem de avaliação como GO e PROG. Ao contrário dos programas LISP, os programas Hope são livres de efeitos colaterais e possuem a propriedade matemática de transparência referencial.

Você deve conhecer os recursos primitivos ou ausentes no LISP e na maioria das linguagens imperativas. A declaração de dados de Hope permite que você defina tipos de dados complexos sem se preocupar como esses dados são representados e a correspondência de padrões permite que você os decomponha para que seja possível usar tipos de dados abstratos diretamente sem escrever procedimentos de acesso e sem a necessidade de inventar muitos nomes novos. O mecanismo de validação de entrada de dados da linguagem permite que o interpretador verifique se você está usando objetos de dados de maneira correta e consistente, enquanto a ideia de tipos polimórficos impede que a verificação seja muito restritiva e permite que você defina formas de dados comuns com uma única função.

Funções de ordem superior e tipos polimórficos permitem escrever programas que são muito concisos. Os programadores são mais produtivos e seus programas são mais fáceis de entender e raciocinar. A propriedade da transparência referencial melhora ainda mais a capacidade de raciocinar sobre os programas e torna possível transformá-los mecanicamente em programas comprovadamente corretos e mais eficientes no uso do espaço e tempo. Finalmente, a transparência referencial mantém o significado dos programas Hope independentes da ordem em que são avaliados tornando-os ideais para avaliação paralela em máquinas que sejam adequadas a este tipo de processamento. Você verá muito mais sobre Hope e idiomas semelhantes no futuro.

Observação final

As tentativas de contato com o autor do artigo não foram bem sucedidas, seu e-mail de contato não está mais disponível para comunicação - a mensagem foi retornada sem ser entregue. Isso decorre de diversos fatores que não vem ao caso. A tradução foi realizada para que o público lusófono que não possui conhecimento de língua inglesa tenha acesso ao conteúdo publicado. Uma cópia do documento original publicado na revista BYTE encontra-se divulgado abertamente no endereço URL: *http://www.stolyarov.info/static/hope/hope_tut/hope_tut.html*.