Algoritmos

CIC/UFRGS

2006/1

Números Naturais

- Definições de dados recursivas até aqui: listas de tamanho arbitrário
- Números naturais também são de "tamanho" arbitrário
- Não existe limite sobre o quão grande um natural poder ser
- Uma função deve ser capaz de processar qualquer um

Roteiro

- Como descrever números de forma recursiva
- Como desenvolver funções que processam números
- formas alternativas de definição

Definindo Números Naturais

- Normalmente por enumeração: 0, 1, 2, etc
- Matemáticos usam 3 pontos: 0, 1, 2, ...
- Para nós nenhuma das formas acima é suficiente
- Qual seria uma forma completa e auto-contida para a descrição dos números naturais?

Definindo Números Naturais

- A única forma de remover o "etc" e os "..." é com uma definição recursiva
- Eis uma primeira tentativa:
 - 0 é um número natural
 - se n é um número natural, então um mais n também é

Definindo Números Naturais

- Eis como fica no estilo de definições de dados de Scheme:

A natural-number is either

1. 0, ou
 2. `(add1 n)` se `n` se n é um número natural
- Poderíamos ter usado `(+n1)`, mas `add1` lembra que estamos falando dos naturais!

Exemplos de Naturais

- 0
- (add1 0)
- (add1 (add1 0))
- (add1 (add1 (add1 0)))
- (add1 (add1 (add1 (add1 0))))

Semelhanças com Listas

- note a semelhança com a definição de listas
- `cons` constrói listas (começando com `empty`)
- `add1` constrói naturais (começando com 0)
- Números naturais possuem abreviaturas 0, 1, 2, 3...
- Mais adiante veremos abreviaturas para listas também

Semelhanças com Listas

- Função sobre naturais extrai o número usado na construção de um natural
- Função sobre listas extrai a lista que foi usada na construção da lista
- Para listas a extração é feita pela operação `rest`
- Para numeros a extração é feita pela operação `sub1`
- `sub1` satisfaz a lei $(\text{sub1 } (\text{add1 } n)) = n$
- Assim como `rest` satisfaz $(\text{rest } (\text{cons } a\text{-value } a\text{-list})) = a\text{-list}$

Processando Naturais de Tamanho Arbitrário

- Desenvolver função `hellos` que consome um natural n e produz lista com n cópias de de `'hello`

```
;; hellos : N -> lista-de-símbolos
;; cria lista de n copias de 'hello
;; exemplos:
;; (hellos 0) produz empty
;; (hellos 2) produz (cons 'hello (cons 'hello empty))
```

Funções para Naturais

- O projeto de um template segue o método adotado para definições de dados recursivas
- A função `hellos` é uma função condicional com duas cláusulas
- Uma das cláusulas para o caso do natural ser 0
- A outras cláusula para natural diferente de zero

```
(define (hellos n)
  (cond
    [(zero? n) ...]
    [else     ...]))
```

Funções para Naturais

- A definição de dados diz que 0 é atômico e qualquer outro natural é um valor composto que "contém" o predecessor (ao qual foi adicionado 1)
- Se n não for 0, subtraímos 1 de n para obtermos o "componente" de n
- O resultado também é um número natural, ao qual se aplica a própria função

```
(define (hellos n)
  (cond
    [(zero? n) ...]
    [else ... (hellos (sub1 n)) ... ]))
```

Funções para Naturais

- Explorada a definição de dados agora vem a parte criativa
- Assuma que `(zero? n)` avalia para `true`: a resposta é `empty`
- Em geral `(hellos (sub1 n))` produz lista com $n - 1$ cópias de `'hello` (pode produzir `empty` também)
- Para produzir lista com n cópias precisa de um `cons`

```
(define (hellos n)
  (cond
    [(zero? n) empty]
    [else (cons 'hello (hellos (sub1 n)))]))
```

Testes

```
(hellos 0)
```

```
= (cond
   [(zero? 0) empty]
   [else (cons 'hello (hellos (sub1 0)))])
```

```
= (cond
   [true empty]
   [else (cons 'hello (hellos (sub1 0)))])
```

```
= empty
```

Teste

```
(hellos 1)
= (cond
  [(zero? 1) empty]
  [else (cons 'hello (hellos (sub1 1)))]])

= (cond
  [false empty]
  [else (cons 'hello (hellos (sub1 1)))]])

= (cons 'hello (hellos (sub1 1)))

= (cons 'hello (hellos 0))
= (cons 'hello empty)
```

Exercícios

1. Generalizar `hello`s definindo uma função `repeat` que consome um número natural n e um símbolo, e produz uma lista com n ocorrências do símbolo
2. Desenvolva a função `!` que calcula fatorial de um natural

Definição Alternativa de Naturais

- Suponha que queiramos definir a função `prod-a-partir-20` que computado o produto dos números a partir de 20 (exclusive) até um número n (inclusive) que seja maior ou igual a 20
- Poderíamos definir uma função que calculasse $(/ (! n) (! 20))$
- Ou poderíamos refinar a definição de dados e aplicar o método do template baseado nessa definição mais precisa

Definição Alternativa de Naturais

- A entrada para essa função pertence a seguinte coleção de números: 20, 21, 22, ...
- Chegamos, portanto, a seguinte definição de dados:

Um número natural [≥ 20] é ou

1. 20 ou
2. $(add1\ n)$ se n é um número natural [≥ 20].

Definição Alternativa de Naturais

- Projeto baseado na definição de dados de $N[\geq 20]$
- Contrato e exemplos:

```
;; prod- apartir-20:  $N[\geq 20] \rightarrow N$   
;; computar n . (n - 1) . ... . 21 . 1  
;; assumir que n  $\geq 20$   
;; exemplos  
;; (= (prod- apartir-20 21) 21)  
;; (= (prod- apartir-20 22) 462)  
;;  
;; por convenção:  
;; (= (prod- apartir-20 20) 1)
```

Definição Alternativa de Naturais

- Template típico para funções com dados definidos recursivamente com duas cláusulas:

```
(define (prod- apartir-20 n)
  (cond
    [(= n 20) ...]
    [else ... (prod- apartir-20 (sub1 n)) ...]))
```

- Parte criativa:

```
(define (prod- apartit-20 n)
  (cond
    [(= n 20) 1]
    [else (* n (prod- apartir-20 (sub1 n)))]))
```

Exercício

1. Desenvolva a função `produto` semelhante a função anterior mas agora recebe como primeiro argumento o limite (ao invés da função anterior que só considerava números maiores ou iguais a 20)
2. O contrato da função é

```
;; produto: N[limit] N[>= limit] -> N
;; computa n . (n - 1)... . (limit + 1) . 1
;; convenção: (produto n n) produz 1
;; exemplos
;; (= (produto 20 21) 21)
;; (= (produto 20 22) 462)
;; (= (produto 10 13) 1716)
```

Exercícios

- 4 A operação `random` do Scheme consome um natural n maior do que 1 e produz um natural entre 0 e $n-1$. A seguinte função usa `random` para produzir um natural em um dado intervalo

```
;; random-n-m : N N --> N
;; dados naturais n e m produz natural entre n e m
;; Assume: n < m
```

```
(define (random-n-m n m)
  (+ (random (- m n)) n))
```

Desenvolva uma função que consome um número natural e produz uma lista com essa quantidade de números escolhidos de forma randômica entre 20 e 120.