Resposta da Tarefa 3 — Funções com Domínios Infinitos, Autômatos e Expressões Regulares & Loops e Infinitude & Modelos Equivalentes de Computação

Análise e Complexidade de Algoritmos
PPComp — Campus Serra, Ifes

Pablo Simões Nascimento

19 de janeiro de 2022

1 Problema 1

Considere a definição "tradicional"de *autômato finito determinístico* (AFD):

$$M = \langle Q, \sum, \delta, q_0, F \rangle$$

Onde,

- Q - é o conjunto de estados de M;
- $\sum$ - é o alfabeto da cadeia de entrada;
- $\delta : Q \times \sum \rightarrow Q$ - é a função de transição de M;
- $q_0 \in Q$ - é o estado inicial de M;
- $F \subseteq Q$ é o conjunto de estados de aceitação de M.

Sejam dois AFDs $M_1 = \langle Q_1, \sum, \delta_1, q_0, F_1 \rangle$ e $M_2 = \langle Q_2, \sum, \delta_2, p_0, F_2 \rangle$. O produto assíncrono de M_1 e M_2, denotado por $M_1 \odot M_2$, é definido como $M_1 \odot M_2 = \langle Q_\pi, \sum, \delta_\pi, s_0, F_\pi \rangle$, onde:

- $Q_\pi = Q_1 \times Q_2$

- $\delta_\pi((q_i, p_i), \sigma) = (\delta_1(q_i, \sigma), \delta_2(p_i, \sigma))$
- $s_0 = (q_0, p_0)$
- $F_\pi = \{(q_f, p_f) | q_f \in F_1 \text{ ou } p_f \in F_2\}$

Parte I - (10 pts)

Prove ou refute a afirmação: o autômato $M_1 \odot M_2$ reconhece a linguagem $L(M_1) \cup L(M_2)$.

Parte II - (10 pts)

Escreva uma função em Clojure que receba como entra a representação de dois AFDs e retorne o AFD correspondente ao produto assíncrono dos dois. O formato dos AFDs deve seguir o que é mostrado no exemplo abaixo.

```
{:alphabet #{0 1}
 :states #{"q0" "q1" "q2" "q3"}
 :initial "q0"
 :accepting #{"q3"}
 :transitions {["q0" 0] "q1"
               ["q0" 1] "q0"
               ["q1" 0] "q2"
               ["q1" 1] "q0"
               ["q2" 0] "q2"
               ["q2" 1] "q3"
               ["q3" 0] "q3"
               ["q3" 1] "q3"}}
```

Parte III - (10 pts)

Sejam $M_1 = \langle Q_1, \sum, \delta_1, q_0, F_1 \rangle$ e $M_2 = \langle Q_2, \sum, \delta_2, p_0, F_2 \rangle$ dois AFDs completos. Defina a operação produto síncrono de M_1 e M_2 denotado por $M_1 \otimes M_2$, de tal modo que $M_1 \otimes M_2$ reconheça a linguagem $L(M_1) \cap L(M_2)$. Ou então, demonstre que não é possível definir esta operação para autômatos finitos determinísticos.

1.1 Solução - Parte I

Vamos provar que o autômato $M_1 \odot M_2$ reconhece a linguagem $L(M_1) \cup L(M_2)$ por absurdo. Suponha xyz uma cadeia reconhecida pela linguagem $L(M_1) \cup L(M_2)$ que não seja aceita pelo autômato $M_1 \odot M_2$.

Temos dois casos: ou xyz é reconhecida pela linguagem $L(M_1)$ ou pela linguagem $L(M_2)$. Seja xyz reconhecido pela Linguagem $L(M_1)$.

- O autômato inicia em s_0, estado associativo com q_0, inicial de M_1.

- O autômato lê x, aciona a função de transição e muda o estado. A definição do novo estado é em parte formada pela transição que ocorreria em M_1 (δ_1), portanto, continuamos mantendo um equivalente de todas opções de transições que tínhamos em M_1 em $M_1 \odot M_2$. Prosseguindo, o autômato lê y, aciona a função de transição e muda o estado; por fim, lê z, aciona a função de transição e muda o estado parando em um certo estado q_f.

- Pela definição de F_π, sabemos que o estado em que o autômato $M_1 \odot M_2$ parar será também um estado de aceitação, pois q_f era um estado de aceitação em M_1. Portanto, $M_1 \odot M_2$ aceita a cadeia xyz.

Chegamos a um absurdo, pois supomos xyz não ser aceito pelo autômato $M_1 \odot M_2$.

De forma análoga temos o segundo caso em que a cadeia xyz é reconhecida pela linguagem $L(M_2)$:

- O autômato inicia em s_0, estado associativo com p_0, inicial de M_2.

- O autômato lê x, aciona a função de transição e muda o estado. A definição do novo estado é em parte formada pela transição que ocorreria em M_2 (δ_2), portanto, continuamos mantendo um equivalente de todas opções de transições que tínhamos em M_2 em $M_1 \odot M_2$. Prosseguindo, o autômato lê y, aciona a função de transição e muda o estado; por fim, lê z, aciona a função de transição e muda o estado parando em um certo estado p_f.

- Pela definição de F_π, sabemos que o estado em que o autômato $M_1 \odot M_2$ parar será também um estado de aceitação, pois p_f era um estado de aceitação em M_2. Portanto, $M_1 \odot M_2$ aceita a cadeia xyz, o que é um absurdo.

1.2 Solução - Parte II

A solução foi criar funções em clojure responsáveis por construir cada parte do autômato e, por fim, chamar todas em uma função principal e construir o autômato final.

Cada função define sua respectiva parte:

- new-alphabet: recebe dois autômatos e retorna o novo alfabeto. Como o problema especifica que ambos autômatos terão o mesmo alfabeto, basta pegar o alfabeto do primeiro autômato.

- new-states: recebe dois autômatos e retorna o conjunto de estados do produto cartesiano entre eles.

- new-initial: recebe dois autômatos e retorna o par-ordenado que define o estado inicial do novo automato assíncrono.

- new-accepting: recebe dois autômatos e retorna o conjunto de estados aceitáveis do novo automato.

- new-transitions: recebe dois autômatos e retorna o conjunto de transições do novo automato.

- assinc-aut: recebe dois autômatos e define o autômato assíncrono entre eles chamando as definições anteriores.

- renomear-estados: recebe um prefixo que será adicionado ao nome dos estados em todo o automato.

Além disso, foi necessário garantir que não possuam os mesmos nomes nos estados, portanto utilizei funções que renomeiam os estados adicionando um prefixo escolhido no nome de cada estado em todo o autômato. Para renomear os estados de um automato, chamar a função "renomear-estados"informando o autômato e um prefixo a ser concatenado no nome dos estados.

Caso não seja necessário renomear os estados, pois já possuam nomes diferentes, não é necessário chamar a função de "renomear-estados", apenas assinc-aut passando os automatos.

Os códigos fontes podem ser verificados nos arquivos /task3/src/probl1/solucao2.clj e orientações em /task3/src/probl1/readme.md.

1.3 Solução - Parte III

Para definição de um autômato que reconheça a linguagem $L(M_1) \cap L(M_2)$ podemos manter a maioria das definições semelhantes ao produto assíncrono, exceto com relação aos estados de aceitação, que é o que define o reconhecimento da cadeia de entrada efetivamente.

No caso caso do produto assíncrono do exercício 1.1, os estados de aceitação F_π são definidos conforme a construção $F_\pi = \{(q_f, p_f) | q_f \in F_1 \text{ ou } p_f \in F_2\}$. Isso significa que F é o conjunto de pares ordenados onde um dos elementos é um estado de aceitação de M_1 ou de M_2. Outra forma de representar esse conjunto é da seguinte forma $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. Observe que os pares ordenados formados consideram sempre algum estado de aceitação representados em F_1 ou em F_2.

No entanto, se queremos agora representar um autômato que aceite a linguagem de interseção de duas linguagens de M_1 e M_2 precisamos garantir que o par ordenado que definirá o estado de aceitação seja formado por um estado de M_1 que seja aceitável e por um estado de M_2 que também seja aceitável. Ou seja, não basta que um dos elementos seja aceitável em seu autômato de origem, mas os dois o devem ser. Neste caso, o conjunto F pode ser dado por $F = F_1 \times F_2$ e o autômato síncrono $M_1 \otimes M_2$ pode ser definido da seguinte forma:

- $Q_\pi = Q_1 \times Q_2$
- $\delta_\pi((q_i, p_i), \sigma) = (\delta_1(q_i, \sigma), \delta_2(p_i, \sigma))$
- $s_0 = (q_0, p_0)$
- $F_\pi = \{(q_f, p_f) | q_f \in F_1 \text{ e } p_f \in F_2\}$

2 Problema 2 *(15 pts)*

Considere o alfabeto $\sum_3$ definido abaixo:

$$\sum_3 = \left\{ \begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}1\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}, \begin{bmatrix}1\\0\\1\end{bmatrix}, \begin{bmatrix}0\\1\\1\end{bmatrix}, \begin{bmatrix}1\\1\\1\end{bmatrix} \right\}$$

Uma cadeia de símbolos de $\sum_3$ pode ser vista como 3 linhas de 0s e 1s. Considere cada linha como um número binário. Seja β a linguagem definida abaixo:

$$\beta = \{w \in \sum_3{}^\star \mid \text{a última linha de } w \text{ é a soma das duas primeiras}\}$$

Por exemplo, $\begin{bmatrix}0\\0\\1\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}1\\1\\0\end{bmatrix} \in \beta$, mas $\begin{bmatrix}0\\0\\1\end{bmatrix}, \begin{bmatrix}1\\0\\1\end{bmatrix} \notin \beta$.

Mostre que a linguagem β é regular.

2.1 Solução

Para mostrar que uma linguagem é regular basta construirmos um autômato que reconheça a linguagem. Esta é a abordagem adotada para solução deste exercício. Para facilitar, chamaremos os elementos do alfabeto $\sum_3$ conforme abaixo:

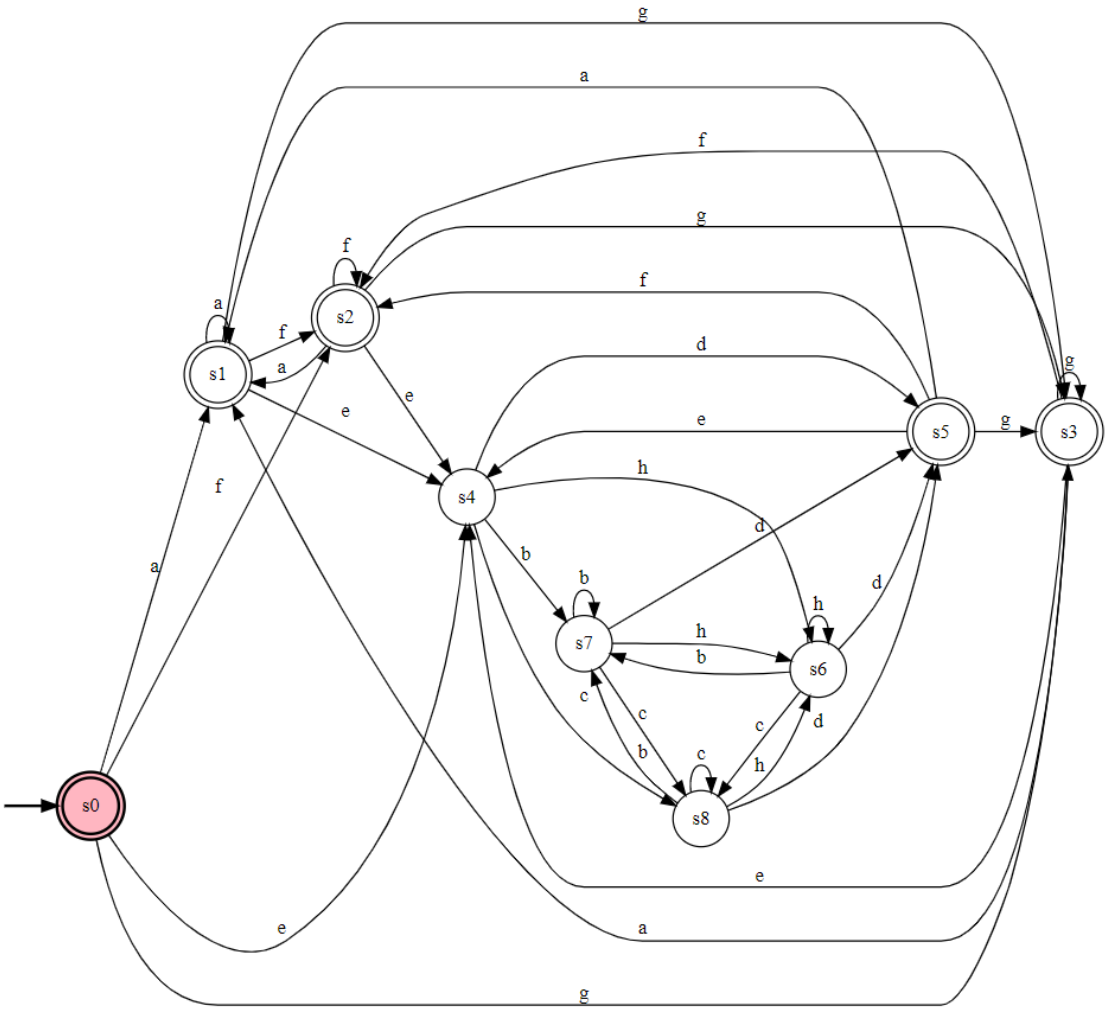
a	b	c	d	e	f	g	h
$\begin{bmatrix}0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\end{bmatrix}$	$\begin{bmatrix}1\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\end{bmatrix}$	$\begin{bmatrix}1\\0\\1\end{bmatrix}$	$\begin{bmatrix}0\\1\\1\end{bmatrix}$	$\begin{bmatrix}1\\1\\1\end{bmatrix}$

O autômato que reconhece a linguagem β pode ser dado conforme Figura 1.
A abordagem para construir o autômato parte do mais simples para o mais complexo:

- Cadeia vazia
- Quais cadeias de apenas um elemento são válidas? a, f e g, pois a condição da soma é válida neles; São, portanto, estados aceitáveis s1, s2 e s3.
- Com quais elementos a sequência não pode começar (visto que falhará na regra da soma)? b, c, d e h. Portanto, restou apenas a letra e, que é o caso quando houver carry na soma, estado s4.
- Caso leia e, o carry será tratado fluindo o autômato pelos estados s4, s7, s6 e s8 e finalizando lendo d e indo para s5.
- todos demais casos serão combinação desse fluxo. Caso chegue em s5, poderá recomeçar tudo novamente lendo a, f, g ou e.

O código fonte de criação do autômato, bem como o link para teste e validação, podem ser verificados nos arquivos /task3/src/probl2/definicao_atomato.txt e /task3/src/probl2/readme.md.

Figura 1: Autômato que reconhece a linguagem β.



3 Problema 3 *(20 pts)*

Para cada uma das funções especificadas abaixo, defina a Máquina de Turing que computa a função. A Máquina de Turing deve ser descrita utilizando a representação definida pelo simulador de Máquina de Turing de Anthony Morphett.

(a) INC - Função que recebe um número binário na fita e produz o seu incremento. O resultado é separado da entrada por um caractere '#'. Por exemplo, para a entrada 10011, após a execução da Máquina de Turing o estado da fita seria 10100#10011.

(b) ADD - Função que recebe dois números binários na fita e produz a sua soma. Os números de entrada e o resultado são separados uns dos outros pelo caractere '#'. Por exemplo, para a entrada 1011#110, o estado da fita após a execução da Máquina de Turing seria 10001#1011#110.

3.1 Solução - Parte (a)

A solução dada foi implementar primeiro uma cópia do número à direita para depois, voltar ao número inicial, e efetuar a soma de um sobre ele.

Portanto, a solução possui 2 partes principais:

- Efetuar a cópia do número separando por um '#';
- Feita a cópia, efetuar o incremento de um no primeiro número à esquerda de '#'.

O código fonte da solução está no arquivo task3/src/probl3/solucao3.1.txt e detalhes no readme.md.

3.2 Solução - Parte (b)

A solução dada foi implementar primeiro uma cópia do número à direita do '#' posicionando-o no início e separando com carcter '='. Em seguida, o último número passa a ser ignorado e apenas os dois primeiros serão lidos e considerados. Da direita para a esquerda, vamos lendo os números e analisando a possibilidade de haver ou não carry (1 + 1).

De maneira geral, a máquina proposta possui 2 partes principais:

- Efetua uma cópia do último número no começo da fita seguido de um sinal =;
- Feita a cópia, inicia-se a soma. Esta soma ignora o último número após o sinal #.

A execução da soma também pode ser compreendida como composta de 3 partes principais:

1. Quando é lido um 1 preparamos o cenário para o caso de um carry;
2. Quando é lido um 0 segue o fluxo conforme a soma com 0 ou 1;
3. Quando finaliza a leitura, os caracteres de representação de dígitos tratados são substituídos novamente para 0s e 1s e para.

O código fonte da solução está no arquivo task3/src/probl3/solucao3.2.txt e detalhes no readme.md.

4 Problema 4 *(15 pts)*

Sejam $\mathbb{0} = \lambda x.\lambda y.y$ e $\mathbb{1} = \lambda x.\lambda y.x$. Defina

$$\text{ALT} = \lambda a, b, c.(a\ (b\ \mathbb{1}(c\ \mathbb{1}\ \mathbb{0}))(b\ c\ \mathbb{0}))$$

Prove que ALT é uma expressão λ que computa a função "pelo menos dois". Ou seja, que para quaisquer $a, b, c \in \{\mathbb{0}, \mathbb{1}\}$ (conforme definidos acima) ALT $a\ b\ c = \mathbb{1}$ se e somente se ao menos duas das entradas forem iguais a $\mathbb{1}$.

4.1 Solução

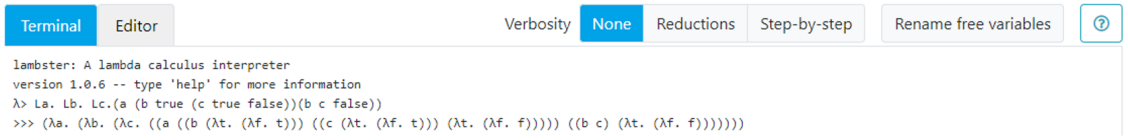
Para solução deste exercício partimos da suposição de que a função ALT comporta-se como um if, que em lambda corresponde justamente a uma expressão com 3 funções "a", "b"e "c"como em $\lambda a, b, c.((a\ b)\ c)$, onde "a"equivale ao teste e caso verdadeiro retorna "b", caso falso retorna "c". Assim, mostraremos que o resultado da execução equivale ao resultado de um if e satisfaz ao comportamento "pelo menos dois".

O ambiente Lambster pode ser utilizado (e foi!) para alguns testes. Por exemplo, observando que o conjunto $\mathbb{0}$ indica falso e que o conjunto $\mathbb{1}$ indica verdadeiro, podemos construir expressões conforme a seguinte ideia:

La. Lb. Lc.(a (b true (c true false))(b c false))

Ao executarmos o comando acima, recebemos a expressão interpretada conforme a Figura 2:

Figura 2: Comando interpretado no ambiente Lambster.



Realizei os testes com todas combinações de entradas possíveis para true e false, conforme Figura 3, e o resultado obtido é mostrado na Figura 4.

Figura 3: Comando para interpretação no ambiente Lambster.

Pelo ambiente temos os resultados e já podemos observar que sim, a função ALT computa o equivalente a "pelo mesno dois". Agora vamos avaliar a expressão passo a passo para tentar identificar o if proposto como equivalente ao ALT e verificar se os resultados são os mesmos.

Considere a expressão abaixo:

$$ALT = \lambda a, b, c.(a\ (b\ \mathbb{1}(c\ \mathbb{1}\ \mathbb{0}))(b\ c\ \mathbb{0})) \tag{1}$$

Avaliando como um if, temos que se "a", então $(b\ \mathbb{1}(c\ \mathbb{1}\ \mathbb{0}))$, caso contrário, $(b\ c\ \mathbb{0}))$.

Vamos avaliar o item mais interno $(c\ \mathbb{1}\ \mathbb{0})$. Com a ajuda do interpretador Lambster, vemos que esta função comporta-se como uma identidade, o argumento que receber é a resposta que retornará, conforme vemos na Figura 5:

Figura 4: Resultado dos comandos interpretados no ambiente Lambster.

Output:

```
>>> (λt. (λf. t))
    ↳ equivalent to: true
>>> (λt. (λf. t))
    ↳ equivalent to: true
>>> (λt. (λf. t))
    ↳ equivalent to: true
>>> (λt. (λf. f))
    ↳ equivalent to: false, zero
>>> (λt. (λf. t))
    ↳ equivalent to: true
>>> (λt. (λf. f))
    ↳ equivalent to: false, zero
>>> (λt. (λf. f))
    ↳ equivalent to: false, zero
>>> (λt. (λf. f))
    ↳ equivalent to: false, zero
```

Figura 5: Avaliação (c 1 0).

```
λ> (Lc.c true false) true
λ > ((λc. ((c true) false)) true)
    δ > expanded 'true' into '(λt. (λf. t))'
    δ > expanded 'false' into '(λt. (λf. f))'
    δ > expanded 'true' into '(λt. (λf. t))'
Δ > ((λc. ((c (λt. (λf. t))) (λt. (λf. f)))) (λt. (λf. t)))
α > ((λc. ((c (λt. (λf. t))) (λt. (λf. f)))) (λX0. (λf. X0)))
β > (((λX0. (λf. X0)) (λt. (λf. t))) (λt. (λf. f)))
β > ((λf. (λt. (λf. t))) (λt. (λf. f)))
β > (λt. (λf. t))
>>> (λt. (λf. t))
    ↳ equivalent to: true

λ> (Lc.c true false) true
>>> (λt. (λf. t))
    ↳ equivalent to: true
λ> (Lc.c true false) false
>>> (λt. (λf. f))
    ↳ equivalent to: false, zero
λ>
```

Ou seja, a avaliação da expressão mantém o valor do argumento que c irá assumir, logo

$(c\ \mathbb{1}\ \mathbb{0}) = c$. Substituindo na expressão resultante, temos:

$$(b\ \mathbb{1}\ (c\ \mathbb{1}\ \mathbb{0})) = (b\ \mathbb{1}\ c)$$

Sabe-se que o operador OR é dado pela expressão $\lambda p.\lambda q.(p\ p\ q)$, logo a expressão $\lambda b.\lambda c.(b\ \mathbb{1}\ c)$ continua equivalente ao OR, pois o termo repetido 'p' recebe $\mathbb{1}$, ou seja, continua avaliando b ou c.

Exemplo: B1C false true = $\lambda p.\lambda q.(p\ \mathbb{1}\ q)(\lambda a.\lambda b.b)(\lambda a.\lambda b.a) = (\lambda a.\lambda b.b)(\lambda a.\lambda b.a)(\lambda a.\lambda b.a) = (\lambda a.\lambda b.a)$ = true. Mesmo o primeiro 'p' sendo false, ainda sim vale o OR. Quando o primeiro 'p' for true, vale o OR também, por definição.

O segundo argumento de ALT é $(b\ c\ \mathbb{0})$. Sabe-se que o operador AND é dado por $\lambda p.\lambda q.(p\ q\ p)$. Semelhante ao caso anterior, neste em particular, quando o segundo 'p' recebe sempre falso, continua valendo o AND.

Exemplo: BC0 true false = $\lambda p.\lambda q.(p\ q\ \mathbb{1}\ )(\lambda a.\lambda b.a)(\lambda a.\lambda b.b) = (\lambda a.\lambda b.a)(\lambda a.\lambda b.b)(\lambda a.\lambda b.a) = (\lambda a.\lambda b.b)$ = true. Em outras palavras se p verdadeiro vale o q, se p falso vale o $\mathbb{1}$, ou seja, verdadeiro.

Assim, a expressão final por ser compreendida como equivalente a "IF (a, OR(b,c), AND(b,c))". Avaliando a tabela verdade deste if temos o seguinte:

a	b	c	or(b,c)	and(b,c)	if(a, or(b,c), and(b,c))
true	true	true	true	true	true
true	true	false	true	false	true
true	false	true	true	false	true
true	false	false	false	false	false
false	true	true	true	true	true
false	true	false	true	false	false
false	false	true	true	false	false
false	false	flase	false	false	false

Tabela 1: Tabela-verdade para expressão if encontrada equivalente a ALT.

Note que a coluna final contendo o if possui os mesmos resultados das entradas conforme testadas inicialmente no ambiente Lambster confirmando nossa suposição inicial. Portanto, provamos que a expressão lambda de ALT computa a função "pelo menos dois", conforme saídas da avaliação if na última coluna da tabela-verdade.

5 Problema 5

Usando a Codificação de Church para Booleanos, números naturais, pares e lista em cálculo λ implemente as funções pedidas abaixo.3

(a) MIN — recebe uma lista de números e retorna o menor deles.

(b) MAX — recebe uma lista de números e retorna o maior deles.

(c) APPEND — recebe duas listas e retorna a concatenação destas duas listas.

(d) REVERSE — recebe uma lista e retorna a lista invertida.

5.1 Solução

Não implementada.