Um Interpretador para uma Linguagem Funcional Orientada por Objetos

Proposta de Projeto de Final de Curso (POC 1)

Área: Linguagens de Programação

Wendell Figueiredo Taveira

Orientadora: Profa. Mariza Andrade da Silva Bigonha

DCC - ICEx - UFMG

Belo Horizonte, 13 de novembro de 1998

1 Motivação

Um programa escrito no paradigma funcional é um conjunto de expressões, funções e declarações que podem chamar-se umas às outras ou usar como argumento o resultado de uma outra função ou expressão. Como o mapeamento dos valores de entrada para os valores de saída é feito mais diretamente, através da construção e aplicação de funções, o paradigma funcional permite construir programas muito simples, concisos, flexíveis e poderosos [10, 11].

É importante para qualquer linguagem de programação possuir um formalismo para definição formal de sua semântica, tanto sob o ponto de vista do projetista de processadores de linguagens, quanto sob o ponto de vista dos programadores que precisam ter um completo entendimento das construções da linguagem [6]. Sob este aspecto, linguagens funcionais se sobressaem em relação às linguagens de outros paradigmas, como por exemplo, o paradigma imperativo [9].

Considerando, portanto, que linguagens funcionais apresentam o formalismo adequado de suas construções, é importante prover aos programadores um sistema eficiente e confiável, no qual possam escrever os programas dentro da especificação da linguagem e gerar o código correspondente ao programa. Qualquer sistema que permita processar programas, executá-los ou prepará-los para execução, é chamado de *processadores de linguagens,* entre os quais citam-se compiladores e interpretadores.

Um compilador é um programa que traduz um programa escrito em uma linguagem de alto nível para uma linguagem intermediária ou código de máquina executável. A característica principal de um compilador é que o programa todo deve ser traduzido antes que se inicie a execução e produza resultados.

Um interpretador é um programa que aceita qualquer programa (o programa fonte) expresso em uma linguagem (a linguagem fonte) e o executa imediatamente. Ele funciona da seguinte forma: ele carrega, analisa e executa as instruções do programa fonte, uma a uma. O programa fonte inicializa a execução e produz resultados assim que a primeira instrução é analisada. Neste

trabalho, a ênfase será dada a interpretadores.

Tradicionalmente, na compilação de linguagens funcionais usa-se como código intermediário a linguagem do Cálculo-Lambda [1] com algumas construções que facilitam a especificação semântica. Neste trabalho estamos trabalhando com $\mathcal{LAMB}$ [8, 9], que corresponde a versão estendida do Cálculo-Lambda. É uma linguagem funcional, de poucas construções sintáticas e uma semântica de fácil entendimento, mas com poder de expressar todos os programas funcionais. $\mathcal{LAMB}$ apresenta como principais características, a existência de um operador capaz de efetuar um *pattern-matching* entre um valor qualquer e um padrão estrutural dado, e a utilização do mecanismo *call-by-need, lazy evaluation* ou *delay rule* para passagem de parâmetros [4].

Especificamente, nossa proposta consiste na interpretação de programas escritos em $\mathcal{LAMB}$ de tal forma que programas escritos em qualquer linguagem e compilados para a linguagem $\mathcal{LAMB}$ possam ser executados.

2 Objetivo

O objetivo do projeto proposto é implementar um interpretador para $\mathcal{LAMB}$. Como subproduto, esse projeto dará ao aluno a possibilidade de desenvolver estudos e trabalhos que irão contribuir para sua formação, além de ser uma proposta com importantes aplicações na área de Linguagens de Programação.

3 Contexto do Trabalho

Este trabalho está inserido no escopo de um projeto que é a compilação de programas em $\mathcal{SCRIPT}$ [2, 9] para código executável, desenvolvido no Departamento de Ciência da Computação. $\mathcal{SCRIPT}$ é uma linguagem funcional orientada por objetos, que visa prover uma notação adequada e estruturada de tal forma a permitir que as descrições de semântica denotacional possam ser efetivamente executadas e depuradas. Foi criada em 1995, em um projeto do professor Roberto S. Bigonha. Apresenta algumas extensões em relação aos elementos básicos de uma linguagem funcional pura, tais como: controle de visibilidade, encapsulação, herança, polimorfismo e ligação dinâmica (*dynamic binding*), o que lhe confere característica de orientação a objetos.

A compilação de $\mathcal{SCRIPT}$ por ser muito grande e complexa foi dividida em várias etapas, a saber: o *Front-End*, o *Back-End* e uma fase intermediária, o $\lambda-$*lifting*. A primeira etapa, o *Front-End* [9], consiste na tradução da linguagem fonte, $\mathcal{SCRIPT}$, para a linguagem intermediária, $\mathcal{LAMB}$. Compreende as fases de análise léxica, análise sintática, criação e gerência da tabela de símbolos, verificação de tipos e geração de código intermediário. A segunda etapa, o *Back-End* [3], consiste na compilação de supercombinadores para código em linguagem C [7]. A terceira etapa, *Lifting*, constitui a etapa intermediária entre o *Front-End* e o *Back-End* do compilador de $\mathcal{SCRIPT}$. Ela consiste na realização do $\lambda-$*lifting*, a tradução de expressões lambda para supercombinadores [5] e na implementação de otimizações necessárias e viáveis. Esta etapa do trabalho está sendo desenvolvida. As outras duas estão completamente implementadas e operacionais.

O projeto que estamos propondo tem por objetivo o desenvolvimento de uma abordagem alternativa para execução do código resultante do compilador de $\mathcal{SCRIPT}$. Em vez de se traduzir o código $\mathcal{LAMB}$ resultante para a coleção de supercombinadores e, posteriormente,

para código C, será feita a execução direta via a interpretação de $\mathcal{LAMB}$, permitindo com isso, maior agilidade na execução dos programas gerados pelo *Front-End* de $\mathcal{SCRIPT}$.

O interpretador $\mathcal{LAMB}$ a ser implementado deverá ser capaz de efetuar a leitura e impressão de expressões $\mathcal{LAMB}$. Para a leitura, será utilizado um analisador LALR(1) [12]. Além disso, o interpretador deverá realizar o *pattern-matching,* e posteriormente, a redução das expressões para um padrão estrutural.

O interpretador $\mathcal{LAMB}$ será implementado na linguagem C e utilizará a versão do $\mathcal{LAMB}$ apresentado por Fabíola Oliveira [9].

4 Plano de Trabalho

Para atingir nosso objetivo, produzir um interpretador para a linguagem $\mathcal{LAMB}$, este projeto foi dividido em três fases. A primeira fase consistirá no estudo dos conceitos relacionados à programação funcional, com ênfase em *pattern matching*, funções de ordem mais alta e avaliação *lazy*. Também será feito um estudo das linguagens $\mathcal{SCRIPT}$ e $\mathcal{LAMB}$. A segunda fase consistirá na especificação e implementação do interpretador. Fazem parte do projeto também, a elaboração de uma massa de testes e produção dos manuais do usuário e do sistema que correspondem à terceira fase. O manual do sistema será escrito ao longo do desenvolvimento do projeto. Para facilitar a visualização do trabalho, a Seção 4.2 apresenta o cronograma de desenvolvimento do mesmo.

4.1 Descrição das Fases

Fase 1: Revisão da literatura

Etapa 01: estudo dos conceitos e princípios do paradigma de programação funcional [10, 11, 5].

Etapa 02: estudo do Cálculo-Lambda - usado como base para a linguagem $\mathcal{LAMB}$ [9, 8].

Etapa 03: estudo da linguagem $\mathcal{LAMB}$ - linguagem que será interpretada [4, 9].

Etapa 04: estudo da linguagem $\mathcal{SCRIPT}$ [2, 9].

Etapa 05: elaboração do relatório parcial.

Fase 2: Desenvolvimento do interpretador

Etapa 06: especificação do interpretador.

Etapa 07: desenvolvimento e implementação.

Fase 3: Testes e elaboração de manuais

Etapa 08: testes.

Etapa 09: elaboração do manual do usuário.

Etapa 10: elaboração final do manual do sistema - será feita no decorrer do desenvolvimento do projeto.

4.2 Cronograma

Etapa 01: 19 de outubro de 1998 até 28 de outubro de 1998.

Etapa 02: 29 de outubro de 1998 até 06 de novembro de 1998.

Etapa 03: 09 de novembro de 1998 até 18 de novembro de 1998.

Etapa 04: 19 de novembro de 1998 até 27 de novembro de 1998.

Etapa 05: 19 de outubro de 1998 até 15 de janeiro de 1999.

Etapa 06: 30 de novembro de 1998 até 15 de janeiro de 1999.

Etapa 07: 18 de janeiro de 1999 até 28 de maio de 1999.

Etapa 08: 31 de maio de 1999 até 25 de junho de 1999.

Etapa 09: 28 de junho de 1999 até 23 de julho de 1999.

Etapa 10: 30 de novembro de 1998 até 23 de julho de 1999.

5 Metodologia de Acompanhamento

Na primeira fase do projeto, haverá duas reuniões semanais, de aproximadamente uma hora, com a professora orientadora para acompanhamento dos estudos do aluno, discussão de funcionamento e relato das dificuldades encontradas. Na segunda fase, o aluno desenvolverá de forma independente seu trabalho, reunindo-se uma vez por semana, durante uma hora, para avaliação das etapas elaboradas e discussão daquelas a serem desenvolvidas. Esta metodologia será empregada durante todo o desenvolvimento do trabalho.

6 Desenvolvimento

6.1 Programação Funcional

A principal característica de linguagens funcionais é a noção de expressão. Basicamente, uma expressão, que pode ser composta por subexpressões, é usada para denotar um valor. Este valor pode ser dos tipos: numérico, lógico, caractere, tupla, função, lista ou qualquer novo tipo definido pelo programador.

A avaliação de uma expressão é feita reduzindo-se a expressão em expressões equivalentes mais simples e pela impressão do resultado [10]. Por exemplo, considere o seguinte *script*, que consiste em uma coleção de uma ou mais equações:

```
dobro x = 2 × x (script)
```

Um seqüência possível de redução é como se segue:

```
dobro (5 + 3) = dobro 8 (+)
              = 2 × 8 (dobro)
              = 16 (×)
```

Neste exemplo, `(+)` e `(×)` referem-se ao uso das *built-ins* de adição e multiplicações, respectivamente, e `(dobro)` refere-se ao uso da regra definida pelo *script* mostrado.

O valor de uma expressão pode também ser definido por análise de condições:

```
max x y = x, se x < y
        = y, se x ≥ y
```

Cada uma das expressões lógicas que distinguem as duas expressões de **max** é denominada *guard*.

A seguir, serão apresentados alguns conceitos que são de grande importância em linguagens de programação funcional.

6.1.1 Valores e Tipos

A maioria das linguagens funcionais fornece tipos similares a outras linguagens. São eles:

NÚMEROS:

consistem de constantes inteiras e constantes fracionárias. Exemplos de constantes numéricas: `35 0 -16.4151`.

Com a utilização de operadores aritméticos como + (adição) e − (subtração) pode-se compor expressões numéricas:

```
?4 + 5
9

?4 - 3
1
```

LÓGICOS:

consistem de duas expressões canônicas que denotam valores-verdades, denominados *True* e *False* e são amplamente utilizados como resultado de comparação, tal como:

```
?4 > 3 + 0
False
```

CARACTERES e *STRINGS*:

consistem de caracteres ASCII. São exemplos de caracteres: `'a'`, `'7'`. Uma seqüência de caracteres constitui uma *string*. São exemplos de *strings*: `"a7"`, `"a"`, `"alô mundo"`.

TUPLAS:

consistem de combinações de tipos para formar novos tipos. Por exemplo, o tipo (num, bool) consiste de todos os pares de valores cujo primeiro componente é um número e o segundo, um valor lógico, como por exemplo: (4, *true*).

6.1.2 Funções de Ordem mais Alta

Uma característica importante de linguagens funcionais é não impor restrições aos tipos dos valores de entrada e retorno de funções, permitindo que as mesmas recebam argumentos de qualquer tipo. Em particular, estes valores podem ser funções. Uma função que recebe uma função como argumento ou retorna uma função como resultado é chamada de função de ordem mais alta [10]. Em outras palavras, linguagens funcionais consideram funções como valores de primeira classe. A principal utilidade de funções de ordem mais alta é tornar o código reutilizável, pela abstração das partes que são específicas a uma aplicação particular [9].

Considere a seguinte definição de **max**:

```
max x y = x, se x < y                    (a)
        = y, se x ≥ y
```

Ao adicionar parênteses aos argumentos de **max**, obtém-se:

```
maxN (x,y) = x, se x < y                 (b)
           = y, se x ≥ y
```

Apesar das funções em (a) e (b) serem bastante parecidas, elas apresentam uma diferença muito importante: elas possuem tipos diferentes. O tipo de **max** é (num) → (num → num), enquanto o tipo de **maxN** é (num × num) → num.

A vantagem de **max** é que ela pode ser chamada somente com um argumento. Por exemplo, (max,2) sempre retorna o argumento caso este seja maior do que 2. Caso contrário, retorna 2.

A função **max** é chamada de versão currificada[1] de **maxN**. Tornar uma função currificada é substituir argumentos estruturados por uma seqüência de argumentos simples. Uma vantagem da versão currificada é que ele reduz o número de parênteses em uma expressão [10].

[1] do inglês, *curried*

6.1.3 Avaliação *lazy*

Avaliação *lazy* é um mecanismo de passagem de parâmetros que faz com que o argumento para uma função seja avaliado na ocasião do seu primeiro uso e não no momento de ativação da função. Se o argumento nunca é usado, então nunca é avaliado. Portanto, é possível passar uma expressão que não pode ser avaliada como argumento para uma função [11].

Para ilustrar, considere a seguinte definição que mostra um exemplo da utilidade da avaliação *lazy*:

```
imprimeA ::  num → char
imprimeA z = 'a'
```

Nesta definição, a função `imprimeA` recebe como parâmetro um valor do tipo numérico e retorna um valor do tipo caractere. Observe que:

```
?imprimeA (10/0)
'a'
```

Repare que na aplicação acima da função imprimeA, o valor `10/0` foi utilizado como argumento para `imprimeA`. No entanto, apesar de `10/0` ser um valor indefinido (por se tratar de uma divisão por zero), a função `dobro` retornou o valor `"a"`. Isto acontece porque como utilizou-se avaliação *lazy*, o valor do argumento de `imprimeA` não é avaliado, e realmente ele não é necessário para determinar o resultado. Diz-se neste caso, que a função é **não estrita**.

6.1.4 Padrões

Pattern matching é um conceito comum em linguagens funcionais modernas, sendo utilizado no contexto de expressões *case*, permitindo definir equações por suas alternativas. Contudo, *pattern matching* é mais geral. Em particular, é possível definir funções usando padrões no lado esquerdo de equações [10]. Por exemplo, considere as seguintes equações:

```
cond True x y = x
cond False x y = y
```

Estas equações podem ser escritas equivalentemente como:

```
cond p x y = x, se p = True
           = y, se p = False
```

Outro exemplo de função definidida por *pattern matching* é:

```
conta 0 = 0
conta 1 = 1
conta (n + 2) = 2
```

Nesta definição, o padrão (n + 2) é associado a um valor, se n casa com um número natural.

De modo geral, uma função pode ser definida por várias equações, cada uma com um lado esquerdo diferente, contendo um padrão (*pattern*) na posição de parâmetro formal. O padrão

deve casar com o argumento da função para a equação poder ser aplicada. Um padrão pode parecer-se com uma expressão, mas cada **identificador livre** (veja Seção 6.2.4) no padrão é **uma ocorrência associada** (veja Seção 6.2.4). O padrão casa com um argumento se a estes identificadores puder ser dado valores que tornam padrão e argumento iguais. Na avaliação do lado direito da equação, os identificadores estão associados a estes valores [11].

6.2 Cálculo-Lambda

A teoria do Cálculo-Lambda [3], desenvolvida em 1930 por Alonzo Church, é um formalismo que fornece regras para manipular funções de uma maneira puramente sintática, podendo ser inserida tanto na teoria lógica da matemática, quanto na teoria de linguagens de programação.

A importância da teoria do Cálculo-Lambda para o estudo da semântica formal de linguagens de programação pode ser explicada pelo fato de que o Cálculo-Lambda permite representar todas as funções computáveis, sem contudo, apresentar sintaxe e semântica complicadas. Neste sentido, linguagens de programação funcionais, que estão diretamente relacionadas ao conceito matemático de funções, podem ser analisadas sob o aspecto da sintaxe e semântica no contexto do Cálculo-Lambda. A seguir, será apresentada uma breve discussão sobre os principais conceitos do Cálculo-Lambda.

6.2.1 Notação

Considere novamente a função **dobro** definida na Seção 6.1:

```
dobro x = 2 × x
```

Esta função poderia ser escrita em Cálculo-Lambda de maneira anônima, da seguinte forma:

λ x.2 × x

A expressão representada por λ x.2 × x é dita ser o valor associado ao identificador **dobro**.

Outro aspecto relevante da notação do Cálculo-Lambda refere-se ao número e à ordem dos parâmetros para as funções. Os parâmetros devem ser especificados entre o símbolo lambda e a expressão que indica o corpo da função. Desta maneira, em uma função especificada por λ y.λ x.x^4 + y^3, o primeiro argumento estará associado a y e o segundo argumento, a x, evitando possíveis ambiguidades.

6.2.2 Sintaxe de Expressões Lambda

O Cálculo-Lambda deriva toda sua utilidade do fato de possuir uma sintaxe esparsa e uma semântica simples e ainda poder representar todas as funções computáveis [6].

A especificação da sintaxe do Cálculo-Lambda na forma **BNF** reflete sua simplicidade:

```
<expressão> ::= <variável>                          ;identificadores em
                                                    letras minúsculas
              |<constante>                          ;objetos pré-definidos
              |(<expressão> <expressão>)            ;combinações
              |(< λ <variável>.<expressão>)         ;abstrações
```

Alguns exemplos de expressões do Cálculo-Lambda são:

1. A expressão λ x.x denota a função identidade, pois $(\lambda$ x.x) E) = E, ou seja, a aplicação da abstração $(\lambda$ x.x) é a função identidade do conjunto dos inteiros, do conjunto de funções do mesmo tipo, ou de qualquer outro tipo de objeto, refletindo o caráter polimórfico do Cálculo-Lambda.

2. A abstração (λ f.(f(f(f x))))) descreve uma função com dois argumentos, uma função e um valor, que aplica a função ao valor três vezes. Considerando a função **dobro** definida na Seção 6.2.1, temos:

```
((( λ f.( λ x.(f(f(f x)))))dobro)2)  = (( λ x.(dobro(dobro(dobro x))))2)
                                     = (dobro(dobro(dobro 2)))
                                     = (dobro(dobro 4))
                                     = (dobro 8)
                                     = 16
```

Algumas convenções foram criadas para reduzir a complexidade gerada pelo número excessivo de parênteses.

1. A aplicação de funções associa-se da esquerda para a direita:

   E_1 E_2 E_3 significa $((E_1\ E_2)\ E_3)$

2. O escopo de "λ <variável>" em uma abstração estende-se o máximo possível para a direita:

   λ x.E_1 E_2 E_3 significa (λ x.(E_1 E_2 E_3)) e não ((λ x.E_1 E_2) E_3)

   Repare que em (λ x.E_1 E_2) E_3, E_3 é um argumento para a função λ x.E_1 E_2 e não parte da função como em (λ x.(E_1 E_2 E_3)). Nos casos em que E_3 for um argumento, o uso dos parênteses é necessário, pois uma aplicação tem precedência maior do que a abstração.

3. Uma abstração permite uma lista de variáveis para abreviar uma série de abstrações lambda:

   λ x y z.E significa (λ x(λ y(λ z.E)))

4. Expressões lambda podem ser nomeadas pelo uso da sintaxe:

   define <nome> = <expressão>

   Por exemplo, dado *define* subtração = λ x.λ y.x−y, segue-se que

   subtração (subtração 9 4) 3) = 2.

De maneira geral, pode-se imaginar que `subtração` é substituído pela sua definição, antes da redução da expressão lambda. As técnicas de redução de expressões lambda são apresentadas na Seção 6.2.6.

6.2.3 Funções Currificadas

Na Seção 6.1.2, foi apresentado o conceito de função currificada. Cálculo-Lambda provê um mecanismo que permite implementar essas funções.

Considere duas versões para uma função que efetua a soma de números naturais:

1. Permitindo pares ordenados como expressões lambda e usando a notação <x,y>, a função **soma** é definida como:

   soma `<a,b> = a + b` (c)

2. Usando a versão currificada da função com a propriedade de que argumentos são fornecidos um de cada vez:

   adição: `(N) → (N → N)` (d)
   onde **adição** `a b = a + b`

   É possível definir a função sucessor como (**adição** 1).

As operações de *currying* e *uncurrying* uma função são expressas em Cálculo-Lambda como:

define Curry = λ f.λ x.λ y.F <x,y>
define Uncurry = λ f . λ p . f (head p) (tail p)

Portanto, as duas versões de operações da adição definidas em `(c)` e `(d)` estão relacionadas entre si da seguinte forma:

`Curry soma = adição` e `Uncurry adição = soma`

6.2.4 Semântica de Expressões Lambda

A avaliação de uma expressão lambda é chamada de redução e retorna uma outra expressão lambda resultante das aplicações das funções.

Dizemos que a ocorrência de uma variável v em uma expressão lambda é **associada** se ela está no escopo de um "λ v". Caso contrário, a ocorrência é dita **livre**. A regra de redução básica envolve a substituição de expressões de variáveis livres similarmente ao modo como parâmetros na definição de uma função são passados como argumentos em uma chamada de função. Portanto, o processo de substituição de variáveis em uma expressão consiste em substituir cada ocorrência de uma variável v em uma expressão E por uma expressão lambda E_1. Indica-se este processo por $E[v \rightarrow E_1]$.

O processo de substituição pode levar a casos em que a semântica das expressões lambda são alteradas. Por exemplo, a substituição (λ x.(mul x y)) [y → x] leva-nos a (λ x.(mul x x)). Repare

que o significado da expressão original era realizar a multiplicação de dois números, enquanto a segunda expressão calcula o quadrado de um número. Isto ocorre porque a substituição efetuada envolve uma **variável capturada**, ou seja, a variável livre x em $[y \rightarrow x]$ tornou-se associada ao resultado da expressão. Uma substituição que não envolve variável capturada é chamada de substituição **válida** ou **segura**.

É necessário portanto, distinguir entre variáveis capturadas e livres em uma expressão lambda. Podemos definir o conjunto de variáveis livres em uma expressão E, denotado por **FV(E)**, como se segue:

a. $FV(c) = \{\ \}$, para qualquer constante c

b. $FV(x) = \{x\}$, para qualquer variável x

c. $FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$

d. $FV(\lambda\ x.E) = FV(E) - \{x\}$

Uma expressão lambda sem variáveis livres ($FV(E) = \{\ \}$) é dita **fechada**. Define-se, então, a substituição de uma variável (livre) em uma expressão como se segue:

a. $v[v \rightarrow E_1] = E_1$, para qualquer variável x

b. $x[v \rightarrow E_1] = x$, para qualquer variável x, $x \neq y$

c. $c[v \rightarrow E_1] = c$, para qualquer constante c

d. $(E_{rator}\ E_{rand})\ [v \rightarrow E_1] = ((E_{rator}\ [v \rightarrow E_1])\ (E_{rand}\ [v \rightarrow E_1]))$

e. $(\lambda\ v.E)\ [v \rightarrow E_1] = (\lambda\ v.E)$

f. $(\lambda\ x.E)\ [v \rightarrow E_1] = (\lambda\ x.(E[v \rightarrow E_1])$, quando $x \neq v$ e $x \notin FV(E_1)$

g. $(\lambda\ x.E)\ [v \rightarrow E_1] = \lambda\ z.(E[x \rightarrow z]\ [v \rightarrow E_1])$, quando $x \neq v$ e $x \in FV\ (E_1)$, onde $z \neq v$ e $z \notin FV(EE_1)$

No item d), E_{rator} refere-se ao operador da combinação (do inglês, *operator*) e E_{rand}, ao operando ou argumento da combinação (do inglês, *operand*).

6.2.5 Tipos de Redução Lambda

A seguir, apresenta-se quatro tipos de redução lambda, onde cada um deles é utilizado em uma etapa no processo de simplificação ou avaliação de uma expressão lambda.

REDUÇÃO-α:
sejam v e w variáveis e E uma expressão lambda. Denotamos a redução-α por:

λ v.E $\Rightarrow$ w.E [v $\rightarrow$ w], desde que w não ocorra em E.

Um exemplo de redução-α é:

λ `z.(`λ `f.fx)z` $\Rightarrow$ λ `z.(`λ `g.gx)z`

REDUÇÃO-β:
sejam v uma variável e E e E_1 expressões lambda. Denotamos a redução-β por:

(λ v.E) E_1 $\Rightarrow$ E [v $\rightarrow$ E_1], desde que as *regras de subsituições seguras* sejam satisfeitas.

A redução-β é a principal regra de simplificação de uma expressão lambda e abrange a operação de aplicação de função.

Uma relação de igualdade entre expressões lambda é definida pelo reverso da redução-β, produzindo a regra de abstração-β. Denota-se por:

E[v $\rightarrow$ E_1] $\Rightarrow$ (λ v . E) E_1

As duas regras, redução-β e abstração-β, constituem a conversão-β, denotada por $\Leftrightarrow$. Desta maneira, duas expressões lambda E e F são iguais se e somente se E $\Rightarrow$ F e F $\Rightarrow$ E, ou de maneira simplificada, E $\Leftrightarrow$ F.

REDUÇÃO-η:
sejam v uma variável e E uma expressão lambda. Denotamos a redução-η por:

λ v.(Ev) $\Rightarrow$ E, desde que v não tenha ocorrência livre em E.

REDUÇÃO-δ:
se o Lambda-Cálculo possui constantes pré-definidas, isto é, não é puro, então as regras associadas aos valores e funções pré-definidos são chamados de regras-δ.

Um exemplo de regra-δ é:

`(add 4 7)` $\Rightarrow$ `11`

6.2.6 Estratégias de Redução e Passagem de Parâmetros

Ao se manipular uma expressão lambda a fim de reduzi-la a uma forma mais simples, é desejável obter uma forma que dê o valor daquela expressão. Assim sendo, dizemos que uma expressão

lambda está na **forma normal** se ela não contém *β-redexes*[2], ou seja, não há expressões que possam ser β-reduzidas.

É importante ressaltar que nem toda expressão lambda pode ser reduzida a uma forma normal. Contudo, caso a redução seja possível, pode haver mais de uma maneira de efetuá-la.

A fim de auxiliar no processo de avaliação de uma expressão lambda, define-se duas estratégias de redução, a saber:

REDUÇÃO DE ORDEM NORMAL:
primeiro reduz o lado esquerdo mais externo da *β-redex* (ou *δ-redex*).

REDUÇÃO DE ORDEM APLICATIVA:
primeiro reduz o lado esquerdo mais interno da *β-redex* (ou da *δ-redex*).

Note que nas definições aparecem os termos "mais externos" e "mais internos" de uma expressão lambda. Assim, para qualquer expressão lambda da forma E = ((λ x . B) A), dizemos que a *β-redex* E é **externa** a qualquer *β-redex* que ocorra em A ou B e que estas são **interna** a E. Uma β-redex em uma expressão lambda está **mais externa** se não há *β-redex* externa a ela, e está **mais interna** se não há *β-redex* interna a ela.

Uma das grandes utilidades das duas estratégias de redução (ordens normal e aplicativa) é permitir simular mecanismos de passagem de parâmetros para procedimentos ou funções em uma linguagem de programação.

Vejamos então a correlação entre as estratégias de redução e a passagem de parâmetros considerando a abstração λ `x.B`, onde `x` é um parâmetro formal e cujo corpo é a expressão lambda `B`. Existem dois mecanismos:

CHAMADA POR NOME:
relaciona-se com a **redução de ordem normal** com a diferença que nenhuma *redex* em uma expressão lambda que esteja no corpo da abstração é reduzida.

Na chamada por nome, um parâmetro real é passado como uma expressão não avaliada. Este parâmetro é avaliado no corpo da função sendo executada toda vez que o parâmetro formal correspondente for referenciado.

CHAMADA POR VALOR:
relaciona-se com a **redução de ordem aplicativa** com a diferença que nehuma *redex* em uma expressão lambda que esteja na abstração é reduzida.

Esta restrição corresponde ao princípio de que o corpo da função não é avaliado até que a função seja chamada em uma redução-β. A ordem aplicativa garante que o argumento para uma função seja avaliado antes dela ser aplicada.

6.3 A Linguagem LAMB

LAMB é uma linguagem funcional que apresenta, portanto, características como avaliação *lazy*, funções de ordem mais alta e casamento de padrões. É utilizada para representar funções matemáticas tais como as funções de definições de semântica denotacional. De modo geral, *LAMB* é usada como linguagem objeto para as linguagens funcionais [3]. A versão de *LAMB* aqui apresentada é a mesma utilizada por Oliveira [9] como linguagem intermediária na compilação de programas escritos em *SCRIPT*.

[2] o termo *β-redex* deriva do termo **red***uction* **ex***pression* (redução de expressão).

6.3.1 Estruturas Básicas da Linguagem

Um programa $\mathcal{LAMB}$ é composto por expressões $\mathcal{LAMB}$ e o objetivo final da execução de um programa consiste em reduzir, **sempre que possível**, as expressões a uma forma mais simples ou **forma normal**, de maneira similar ao descrito na Seção 6.2.6.

A seguir, apresentamos as características básicas da linguagem, extraído de Oliveira [9] e Maia [3].

- As constantes de $\mathcal{LAMB}$ são dos seguintes tipos:
  - **números inteiros:** são escritos na forma decimal e estão no domínio dos inteiros, representado por $\mathcal{N}$. Exemplo: $\ldots, -2, -1, 0, 1, 2, \ldots$.
  - **quotations:** são seqüências de caracteres entre aspas e estão no domínio das *quotations*, representado por $\mathcal{Q}$. Exemplo: "Lamb", "H2O/1".
  - **valores lógicos:** são representados por TT (verdadeiro) e FF (falso) e estão no domínio dos valores booleanos, representados por $\mathcal{T}$.
  - **elemento ?:** constante que indica valor indefinido. É usado para denotar valores de expressões sem valor, como erros semânticos.
- As variáveis de $\mathcal{LAMB}$ são representadas por identificadores que são formados por letras minúsculas, hífens e dígitos decimais.
- Listas de tamanho finito indeterminado são designadas escrevendo-se os elementos constantes separados por vírgulas entre < e >, <const$_1$, ..., const$_n$> e estão contidas no domínio $\mathcal{L}$. Exemplo: <25,"POC", TT, FF, ?, "FIM">.
- Tuplas finitas são designadas escrevendo-se os elementos constantes, listas ou funções separados por vírgulas entre (e), (const$_1$, ..., const$_n$) e estão no domínio $\mathcal{C}$. Exemplo: (TT, <"1", TT>).

6.3.2 Operadores

Os operadores de $\mathcal{LAMB}$ podem ser divididos nos seguintes tipos:

- **Operadores para listas:** considere e, e$_1$, ..., e$_n$ como expressões $\mathcal{LAMB}$ arbitrárias. Os operadores existentes para manipular listas são:
  - <e$_1$, ..., e$_n$>: cria uma lista com os componentes e$_1$, ..., e$_n$.
  - SIZE e: número de elementos da lista e.
  - e$_1$ EL n: n-ésimo elemento da lista e$_1$ ou ? caso n seja maior do que SIZE e$_1$.
  - e$_1$ CAT e$_2$: concatenação das listas e$_1$ e e$_2$.
  - CONC <e$_1$, ..., e$_n$>: cria uma lista formada através da concatenação (CAT) das listas e$_1$, ..., e$_n$. Equivale a e$_1$ CAT ... CAT e$_n$.
  - e$_1$ AUG e: concatena a lista <e> à última posição da lista e$_1$. Equivale a e$_1$ CAT <e>.
  - e PRE e$_2$: concatena a lista e$_2$ à última posição da lista <e>. Equivale a <e> CAT e$_2$.
  - HEAD e: cabeça da lista e.

- TAIL e: cauda da lista e.

- **Operadores aritméticos:** considere $n_1 \in \mathcal{N}$ e $n_2 \in \mathcal{N}$. Com os operadores binários infixados PLUS, MINUS, MULT, DIV, REM podemos formar as seguintes expressões $\mathcal{LAMB}$ cujos valores também estão no domínio $\mathcal{N}$:

  - n_1 PLUS n_2: soma.
  - n_1 MINUS n_2: subtração.
  - n_1 MULT n_2: multiplicação.
  - n_1 DIV n_2: divisão; se $n_2 = 0$, o valor da expressão é ?.
  - n_1 REM n_2: resto da divisão; se $n_2 = 0$, o valor da expressão é ?.

- **Operadores relacionais:** considere n_1 e n_2 como expressões nos domínios $\mathcal{N}$ ou $\mathcal{Q}$. Com os operadores binários infixados LS, LE, GR, GE podemos formar as seguintes expressões $\mathcal{LAMB}$ cujos valores são TT se n_1 e n_2 satisfizerem as relações correspondentes ou FF caso não satisfizerem:

  - n_1 LS n_2: (lexicograficamente) menor.
  - n_1 LE n_2: (lexicograficamente) menor ou igual.
  - n_1 GR n_2: (lexicograficamente) maior.
  - n_1 GE n_2: (lexicograficamente) maior ou igual.

- **Operadores lógicos:** considere t_1 e t_2 como expressões no domínio $\mathcal{T}$. Com os operadores binários infixados AND e OR e com o operador unário prefixado NOT podemos formar as seguintes expressões $\mathcal{LAMB}$ cujos valores são TT se t_1 e t_2 satisfizerem as relações correspondentes ou FF caso não satisfizerem:

  - t_1 AND t_2: valor lógico *e*.
  - t_1 OR t_2: valor lógico *ou inclusive*.
  - NOT t_1: negação do valor lógico t_1.

- **Operadores condicionais:** sejam e, e_1 e e_2 expressões $\mathcal{LAMB}$ e p um padrão. Dispomos dos operadores EQ, NE e IS para testar a igualdade de expressões.

  - e_1 EQ e_2: retorna TT se e_1 e e_2 têm o mesmo valor; retorna FF se e_1 e e_2 são valores funcionais ou se têm valores distintos.
  - e_1 NE e_2: representa a negação da expressão e_1 EQ e_2.
  - e_1 IS p: testa estrutura (forma) da expressão e_1 e do padrão p.

- **Operadores para cadeias de caracteres:** sejam q, q_1 e q_2 elementos pertencentes ao domínio $\mathcal{Q}$. Com os operadores NUMBER, QUOTE, TRUTH e CAT podemos realizar a conversão de listas de caracteres em números inteiros, *quotations* ou valores lógicos, e também concatenar duas *quotations*.

  - NUMBER q: converte a lista de caracteres em um número inteiro. Exemplos:

```
NUMBER <"9", "5", "0"> = 950
NUMBER <> = ?
```

- QUOTE q: converte a lista de caracteres em uma *quotation*. Exemplos:

```
QUOTE <"T", "F"> = "TF"
QUOTE <> = " "
```

- TRUTH q: converte a lista de caracteres em um valor lógico. Exemplos:

```
TRUTH <"F", "F"> = FF
TRUTH <"T", "F"> = ?
```

- q_1 CAT q_2: concatena as *quotations* q_1 e q_2. Exemplo:

```
"ab" CAT "cd" = "abcd"
```

- **Operadores para tuplas:** sejam e, e_1, ..., e_n expressões $\mathcal{LAMB}$ e e_1' e e_2' tuplas. Os operadores que operam sobre tuplas são:
  - $(e_1, \ldots, e_n)$: cria uma tupla com os componentes e_1, ..., e_n.
  - e_1' EXT e_2': concatenação das tuplas e_1' e e_2'.
  - e_1' EL *n*: n-ésimo componente da tupla e_1' ou ? caso *n* seja maior do que o número de componentes de e_1'.
  - e*: denota uma tupla finita com componentes no domínio de e.
  - e+: denota uma tupla finita com pelo menos um componente no domínio de e.

- **Operadores para árvores sintáticas:** um domínio de nodos de árvores sintáticas consiste em um rótulo (*label*) e zero ou mais sub-árvores. O rótulo é denotado por uma *quotation* e as sub-árvores que compõem a árvore são denotadas por uma tupla. Temos os seguintes operadores para construir e manipular árvores sintáticas em $\mathcal{LAMB}$.
  - $[D_1, \ldots, D_n]$: cria um nodo com *i* sub-árvores, $1 \leq i \leq n$.
  - q NODE e: cria um nodo cujo rótulo é a *quotation* q e as sub-árvores são os componentes da tupla e.

6.3.3 Funções

A definição de funções em $\mathcal{LAMB}$ é feita por meio do mecanismo de abstração $\mathcal{LAMB}$ denotado por $\mathcal{LAM}$ `x.e`, que introduz o identificador `x` no escopo da expressão `e`, criando uma abstração funcional. Assim, a função **dobro** definida na Seção 6.1 pode ser escrita em $\mathcal{LAMB}$ como:

```
LAM x.(2 × x)
```

A aplicação de função em $\mathcal{LAMB}$ é designada por justaposição e obedece à precedência natural da esquerda para a direita. Desta maneira, uma aplicação da forma `f; g; e;` representa a chamada da rotina `f` com o argumento sendo a expressão resultante da chamada da rotina `g` com o argumento `e`. Pode também ser expressa como `f g e` ou `(f(g(e)))`.

Define-se também, dois operadores para composição de funções. Os operadores são mostrados a seguir:

- f CIRC g: LAM VAL x . ((LAM VAL x_1 . $g(x_1)$) $f(x)$).
- f STAR g: LAM VAL x . ((LAM VAL <x_1, x_2> . $g(x_1)(x_2)$) $f(x)$).

Os operadores definidos acima foram construídos de forma a terem semântica estrita. Para tal, utilizou-se o operador especial de avaliação VAL que força uma avaliação estrita de uma determinada expressão.

Um outro operador de avaliação, SPECIAL, também é definido e permite que funções declaradas sejam compiladas diretamente para a linguagem C.

Funções recursivas podem ser definidas utilizando uma expressão com o operador de ponto fixo. Essa expressão é do tipo $\mathcal{FIXLAMB}$ `f.e`, e denota o valor de `Y (LAM f.e)`, onde `Y` é combinador paradoxal de Curry [13] definido como `Y = LAM a.(LAM b.a(b(b))) (LAM b.a(b(b)))`.

6.3.4 Definições Independentes de Expressões

É possível gerar expressões com um ambiente local gerado por lista de definições. Existem duas formas de construir tais expressões:

- LET $p_1 = e_1$ ALSO $p_2 = e_2$ ALSO ... IN e, onde p_1, p_2, ... são padrões $\mathcal{LAMB}$ e e_1, e_2, ... são expressões $\mathcal{LAMB}$. Os identificadores que compõem cada padrão são associados com as subexpressões correspondentes, definindo o ambiente no qual a expressão final será avaliada.
- DEF $p_1 = e_1$ WITH $p_2 = e_2$ WITH ... IN e, onde p_1, p_2, ... são padrões $\mathcal{LAMB}$ e e_1, e_2, ... são expressões $\mathcal{LAMB}$. Cada definição é avaliada no ambiente composto por todas as demais associações produzidas pelas outras definições.

6.3.5 Padrões

Um padrão representa uma estrutura de **expressão** e em $\mathcal{LAMB}$ os tipos mais simples são:

- o valor especial ?, "indefinido", que casa com qualquer valor;
- um identificador que representa uma variável que casa com qualquer valor;
- uma constante literal, com exceção de ?, que casa somente com ela mesma;
- uma combinação de padrões mais simples com operadores de padrões (tuplas, nodos e abstrações $\mathcal{LAMB}$).

Existem quatro pontos na linguagem $\mathcal{LAMB}$ onde um padrão pode aparecer:

1. como parâmetro formal de uma abstração $\mathcal{LAMB}$;
2. no lado esquerdo de uma definição de uma construção LET ou DEF;
3. como argumento do operador IS;
4. na composição de um novo padrão.

O casamento de padrões verifica se a estrutura do padrão é a mesma do valor. Em $\mathcal{LAMB}$ os padrões atuam como seletores, ou seja, uma função seletora é capaz de selecionar cada componente do padrão por meio de sua posição no padrão.

Considerando e, e_1, e_2, ..., e_m como expressões $\mathcal{LAMB}$, *id* como um identificador e *n* como uma variável inteira contida em $\mathcal{N}$, podemos construir um novo padrão das seguintes formas:

- $\mathcal{LAM}$?.?: casa com abstrações de funções.
- (e_1, ..., e_m): casa com tuplas com m componentes.
- <e_1, ..., e_m>: casa com listas com m componentes.
- [e_1, ..., e_m]: casa com nodos de árvores sintáticas.
- <e*>: casa com listas de tamanho qualquer.
- <e+>: casa com listas de tamanho qualquer, com pelo menos um componente.
- e_1 PRE e_2*: casa com listas com pelo menos um componente.
- e_1 AUG e_2*: casa com listas com pelo menos um componente.
- *id*[*n*] EXT e ou e_1 EXT e_2: casa com tuplas de tamanho qualquer.
- e_1 NODE e_2: casa com nodos de árvores sintáticas.
- NUMBER e: casa com números.
- TRUTH e: casa com valores booleanos.
- QUOTE e: casa com *quotations*.

6.3.6 Passagem de Parâmetros

A passagem de parâmetros em $\mathcal{LAMB}$ é feita por meio de um mecanismo no qual a avaliação dos argumentos é atrasada até que eles sejam requisitados em uma referência. A este mecanismo dá-se o nome de chamada preguiçosa[3]. $\mathcal{LAMB}$ utiliza especificamente, um tipo de avaliação *lazy* denominada chamada por necessidade[4].

Vejamos como o mecanismo de avaliação *lazy* funciona em $\mathcal{LAMB}$. Seja *exp* o nome de uma expressão que não pode ser reduzida a uma forma normal, conduzindo a um processo interminável quando se tenta avaliá-la. Considere agora a seguinte expressão:

```
(LAM <a, b>.(a REM 2) GE (a DIV 2)) <3, exp>
```

Na avaliação desta expressão, utilizando-se *call by need*, como *b* não é referenciado no corpo da abstração, seu valor, que está associado a exp, não precisa ser calculado.

De maneira geral, a avaliação *lazy* adiciona expressividade à linguagem por possibilitar a construção e manipulação de estruturas de dados infinitas. Dos dois mecanismos de avaliação *lazy* existentes, *call by need* e *call by name*, o primeiro é mais eficiente por associar a cada parâmetro um indicador se ele já foi avaliado ou não. Desse modo, evita-se que o parâmetro seja avaliado toda vez que for referenciado.

[3]do inglês, *call by lazy*
[4]do inglês, *call by need*

sintaxe de Lamb

6.4 A Linguagem SCRIPT

Por motivos de completeza, será apresentada a linguagem $\mathcal{SCRIPT}$, para a qual foi desenvolvido um compilador [9]. No entanto, o interpretador a ser implementado será capaz de executar qualquer código $\mathcal{LAMB}$ independente do processo de geração deste código.

A linguagem $\mathcal{SCRIPT}$ foi desenvolvida por Roberto Bigonha [2], tendo como base a linguagem SDL [8]. Sua principal utilidade é prover um ambiente no qual descrições semânticas possam ser efetivamente executadas e depuradas de maneira estruturada.

Esta seção abordará os principais pontos da descrição da linguagem $\mathcal{SCRIPT}$. Maiores detalhes podem ser obtidos em Bigonha [2] e Oliveira [9].

6.4.1 Domínios da Linguagem

Os domínios de $\mathcal{SCRIPT}$ definidos são os seguintes:

- **Domínio básico dos números inteiros:** constituído pelos inteiros decimais contidos na faixa -32768, ..., 32767 e representado por $\mathcal{N}$.
- **Domínio básico das *strings*:** constituído por seqüências de caracteres `ASCII` delimitadas por aspas e representado por $\mathcal{Q}$.
- **Domínio básico dos valores lógicos:** constituído pelas duas *strings* padrões, TT (verdadeiro) e FF (falso). É representada por $\mathcal{T}$.
- **Domínio básico dos valores indefinidos:** constituído pelo símbolo ?.
- **Domínios constantes:** todas as *quotations* estão contidas no domínio $\mathcal{Q}$. Entretanto, uma *quotation* poderá ocorrer em um local onde um domínio era esperado. Neste caso, a *quotation* representará o domínio cujo único elemento diferente do *bottom*[5] é a própria *quotation* que também dá nome ao domínio. Por exemplo:

  ```
  DOMAINS linguagem = "script";
  ```

  A *quotation* `"script"` e o elemento $\perp$ são os únicos componentes do domínio `"script"`. Linguagem é equivalente a este domínio.
- **Domínio de tuplas:** tuplas são designadas por (a_1:d_1, a_2:d_2, ..., a_n:d_n) onde d_i é o domínio do elemento a_i, para $1 \leq i \leq n$.
- **Domínios de listas:** listas podem ser designadas de três formas diferentes:
  1. d*: representa uma lista finita contendo qualquer número de componentes no domínio d.
  2. d+: representa uma lista finita contendo pelo menos um componente no domínio d.
  3. <a_1, a_2, ..., a_n>: representa uma instância de uma lista com os componentes a_1, a_2, ..., a_n, todos de um mesmo domínio d.
- **Domínio de funções contínuas:** o domínio das funções contínuas de d_1 a d_2 é denotado pela expressão $d_1 \rightarrow d_2$, representando o mapeamento de um domínio d_1 em um domínio d_2.

[5] o valor especial *bottom* ($\perp$) serve para modelar semântica de programas com execução infinita.

- **Domínio de nodos:** os nodos de árvores sintáticas com o mesmo *label* são representados por [D_1, ..., D_n]. O *label*, que serve para distinguir os nodos, é definido implicitamente pela *quotation* QUOTE <q_1, q_2, ..., q_n>, onde $m \leq n$. Cada um dos q_i, $1 \leq i \leq n$, é o nome dos domínios ocorrendo na mesma ordem que em [D_1, ..., D_n].

6.4.2 Expressões

As expressões em *SCRIPT* podem ser expressões básicas, expressões de padrões, expressões de comparação, expressões condicionais, expressões CASE, expressões de abstração, expressões LET, aplicações de funções ou quaisquer outras combinações bem formadas de expressões mais simples com operadores.

A seguir, cada uma das expressões é apresentada em detalhes.

- **Expressões básicas:** são as seguintes expressões básicas de *SCRIPT*.
  - **constantes literais:** valores pertencentes aos domínios básicos dos números inteiros, das *strings*, dos valores lógicos e dos valores indefinidos;
  - **variáveis:** denotam membros de domínios;
  - **inteiras:** formadas pela combinação de números inteiros com os operadores PLUS, MINUS, MULT, DIV, REM e SIZE;
  - *quotations:* formadas pela combinação de *quotations* com os operadores LT, LE, GT, GE, EQ, NE, CAT, QUOTE, NUMBER e TRUTH;
  - **expressões lógicas:** formadas por expressões no domínio $\mathcal{T}$ e pelos operadores AND, OR, NOT, EQ e NE;
  - **expressões de listas:** expressões que possuem os construtores "*", "+", "<",">" e os operadores relacionados CAT, CONC, PRE e AUG;
  - **expressões de tuplas:** expressões envolvidas por parênteses. O operador relacionado a tuplas é EXT;
  - **expressões com nodos:** expressões da forma [e_1, ..., e_n], onde e_i, $1 \leq i \leq n$, é qualquer tipo de expressão.
- **Expressões de padrões:** uma expressão de padrão em *SCRIPT* pode ser:
  - uma constante literal: casa com ela mesma;
  - um identificador: tratado como o valor ? quando aplicado ao operador IS;
  - o valor indefinido ?: casa com valores de qualquer tipo;
  - uma combinação de expressões de padrões mais simples e operadores de construção de padrão.

  Sendo e, e_1, ..., e_n expressões de padrões, podemos construir novos padrões das seguintes formas:

  1. (e_1, ..., e_n): casa com tuplas com n componentes.
  2. e*: casa com listas com qualquer número de componentes.
  3. e+: casa com listas com pelo menos um componente.

4. e_1 PRE e_2*: casa com listas com pelo menos um componente.
5. [e_1, ..., e_n]: casa com nodos.
6. NUMBER e+: casa com números inteiros.
7. TRUTH e+: casa com valores booleanos.
8. QUOTE e*: casa com *quotations*.

- **Expressões de comparação:** sejam e_1 e e_2 expressões em *SCRIPT* e p uma expressão de padrão. Existem três formas distintas de comparar expressões em *SCRIPT*:
  - e_1 EQ e_2: testa se as expressões e_1 e e_2 denotam o mesmo valor. É avaliada como TT se e_1 e e_2 possuírem o mesmo valor e como FF se possuírem valores diferentes ou se pelo menos uma delas possuírem valor funcional.
  - e_1 NE e_2: representa a negação da expressão e_1 EQ e_2.
  - e_1 IS p: testa se a expressão e_1 tem a forma particular (estrutura) do padrão p.

- **Expressões condicionais:** uma expressão condicional em *SCRIPT* é apresentada como `t` $\rightarrow$ e_1, e_2, correspondendo ao caso onde `t` é uma expressão cuja avaliação retorna TT, FF, ? ou $\perp$ e e_1 e e_2 são expressões quaisquer. Caso `t` seja avaliado como TT, e_1 será a expressão correspondente; se `t` for avaliado como FF, e_2 será a expressão correspondente. A expressão correspondente será ? ou $\perp$, caso `t` denote, respectivamente, ? ou $\perp$.

- **Expressões CASE:** uma construção **CASE** é utilizada para pesquisar qual a estrutura ou forma de um valor. Expressões **CASE** são denotadas por uma expressão e por uma série de padrões que produzem como resultado um expressão associada ao primeiro padrão que corresponde à estrutura do valor dado. Portanto, para quaisquer expressões `e`, e_1, ..., e_n e para os padrões p_1, ..., p_n, a expressão **CASE** possui a seguinte forma:

```
CASE e
  p1 → e1
     ...
  pn → en
  END
```

- **Expressões de abstrações:** são as seguintes expressões de abstrações de *SCRIPT*:
  - **abstrações LAM:** a operação `LAM x.e` é utilizada para representar as funções não-recursivas anônimas, sendo que a função do operador LAM é associar o identificador `x` no escopo da expressão `e`.
  - **abstrações de padrões:** a operação `LAM p.e`, com `p` sendo um padrão e `e` uma expressão, permitindo a ocorrência de padrões em expressões do tipo LAM, é utilizada para extrair componentes em um valor.

- **Expressões LET:** a forma geral de uma expressão **LET** é:

$$\texttt{LET } a_1 = e_1 \texttt{ LET } a_1 = e_1 \ldots \texttt{ LET } a_i = e_i \texttt{ IN e}$$

onde temos a_i, $1 \leq i \leq n$, definida no escopo das expressões e_i e e. Cada a_i é dito estar na forma de **ligação de padrões** ou **definição de função**.

- **Aplicações de funções:** em *SCRIPT*, uma expressão como `f g e` com `f` e `g` sendo expressões denotando funções e `e` uma expressão arbitrária, é construída da forma `(f(g))(e)`. Outras maneiras possíveis, que eliminam o uso de parênteses, são `f; g; e` e `f$g(e)`.

Em relação à passagem de parâmetros, *SCRIPT* utiliza o mecanismo de avaliação *lazy* de maneira similar ao descrito na Seção 6.1.3.

Outras características importantes de *SCRIPT* são permitir a definição de funções associadas a domínios de tuplas e sobrecarga (*overloading*) de funções.

Referências

[1] Church, A. *The Calculi of Lambda-Conversion.* Ann. of Math. Studies 6, Princeton University.

[2] Bigonha, Roberto da Silva. *The Revised Report on the SCRIPT Language for Denotational Semantics.* Relatório Técnico 016/97. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, julho, 1997.

[3] Maia, Marcelo de Almeida. *Implementação Eficiente de uma Linguagem para Definição de Semântica Denotacional.* Tese de Mestrado, DCC/UFMG, 1994.

[4] Silva, Luiz Ricardo de Faria Silva. *LAMB - Um Interpretador para o Cálculo-Lambda Estendido.* Anais do X Seminário Integrado de *Software* e *Hardware*, III Congresso da Sociedade Brasileira de Computação, Campinas, 23-29/1983, páginas: 487-499, 1983.

[5] Jones, Simon L. Peyton. *The Implementation of Functional Programming Languages.* C.A.R Hoare Series Editor, 1989.

[6] Slenneger, K. and Kurts, Barry. *Formal Syntax and Semantics of Programming Languages.* Addison-Wesley, 1995.

[7] Kerningham, B. and Ritchie, D. *The C Programming Language.* Prentice Hall, 1988.

[8] Mosses, P. D. *SIS - A Compiler-Generator System Using Denotational Semantics.* Technical Report, University of Aarhus, Denmark, 1978.

[9] Oliveira, Fabíola Fonseca. *Compilação de uma Linguagem Funcional, Orientada por Objetos, para Definição de Semântica Denotacional.* Tese de Mestrado, DCC/UFMG, 1998.

[10] Wadler, Philip and Bird, Richard. *Introduction to Functional Programming.* C.A.R. Hoare Series Editor, 1988.

[11] Watt, David A. *Programming Language Concepts and Paradigms.* C.A.R. Hoare Series Editor, 1989.

[12] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers Principles, Techniques, and Tools.* Addison Wesley, 1986.

[13] Burge, W. H. *Recursive Programming Techniques.* Addison Wesley, Mass, 1975.

Wendell Figueiredo Taveira
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Tel.: (031)499-5860 / (031)499-5842
Fax.: (031) 499-5858

Prof. Mariza Andrade da Silva Bigonha
Professor Adjunto
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Tel.: (031)499-5860 / (031)499-5891
Fax.: (031) 499-5858