Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Relatório do Projeto de Final de Curso (POC II)

Compilador da Linguagem Funcional Orientada por Objetos *SCRIPT* para *HASKELL*

Aluno: Wendell Figueiredo Taveira
Email: wendell_taveira@yahoo.com

Orientadora: Mariza Andrade da Silva Bigonha
Email: mariza@dcc.ufmg.br

Belo Horizonte
30 de julho de 1999

Resumo

Este trabalho mostra a compilação de programas escritos na linguagem *SCRIPT* para *HASKELL*. *SCRIPT* é uma linguagem puramente 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.

Considerando o fato de que existem eficientes compiladores de *HASKELL* disponíveis no mercado, o desafio de desenvolver um compilador eficiente para *SCRIPT* motivou a produção de um compilador de *SCRIPT* para *HASKELL*. Especificamente nosso trabalho consistiu na implementação do *back-end* do mesmo.

Este trabalho está inserido no escopo de um projeto que é a compilação de programas em *SCRIPT* para código executável. Já foram desenvolvidos, por outros membros da equipe do Laboratório de Linguagens de Programação da UFMG, o *front-end* e o *back-end* do compilador de *SCRIPT*, este último, gerando código C.

Sumário

Lista de Figuras

1 Introdução

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 interpretadores e compiladores.

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 inicia a execução e produz resultados assim que a primeira instrução é analisada.

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. Neste trabalho, a ênfase será dada a compiladores.

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. A partir daí, há vários caminhos possíveis para se atingir um código executável. Pode-se, por exemplo, gerar código para máquina SECD [3] ou ainda basear a compilação em combinadores [15]. Entretanto, a obtenção de compiladores eficientes para linguagens funcionais ainda é assunto de estudos e pesquisas e requer muito esforço para alcançar resultados expressivos.

Considerando o fato de que existem eficientes compiladores de $\mathcal{HASKELL}$ disponíveis no mercado, o desafio de desenvolver um compilador eficiente para linguagens funcionais, reduzindo ao máximo o esforço necessário para implementá-lo, motivou a produção de um compilador de $\mathcal{SCRIPT}$ [2, 9] para $\mathcal{HASKELL}$ [14]. O objetivo do nosso trabalho é a implementação desse compilador, mais especificamente o *back-end* do mesmo. A vantagem do trabalho proposto é disponibilizar, de forma rápida e eficiente, um compilador de $\mathcal{SCRIPT}$ sem a necessidade de se implementar a parte do compilador que geraria o código executável, tarefa esta que será executada pelo compilador $\mathcal{HASKELL}$.

Este texto está organizado da seguinte forma:

- A Seção 2 mostra o contexto no qual o projeto está inserido.
- A Seção 3 apresenta a revisão da literatura salientando os seguintes pontos:
  - principais conceitos referentes a linguagens de programação funcional;

- teoria do Cálculo-Lambda que representa um formalismo teórico no desenvolvimento de linguagens funcionais;
- a linguagem $\mathcal{LAMB}$, que era a linguagem para a qual o compilador $\mathcal{SCRIPT}$ originalmente gerava código;
- a linguagem $\mathcal{SCRIPT}$, que é a linguagem para definição de semântica denotacional, para a qual desenvolveu-se um processo de geração de código;
- a linguagem $\mathcal{HASKELL}$, que é a linguagem intermediária para a qual foi gerado código.

- A Seção 4 apresenta a especificação para a geração de código $\mathcal{HASKELL}$, incluindo os esquemas de tradução e as rotinas semânticas.
- A Seção 5 apresenta detalhes de implementação do compilador $\mathcal{SCRIPT}$.

2 Contexto do Trabalho

Este trabalho está inserido no escopo de um projeto que é a compilação de programas em $\mathcal{SCRIPT}$ para código executável, desenvolvido no Departamento de Ciência da Computação. $\mathcal{SCRIPT}$ é uma linguagem puramente 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[1], 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, $\mathcal{LAMB}$ [8, 9]. 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. A Figura 1 apresenta, de forma simplificada, as etapas envolvidas na construção desse compilador.

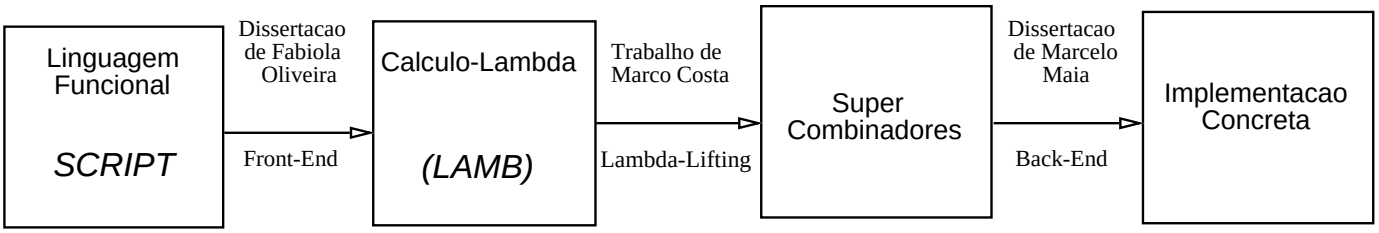

Figura 1: Compilador $\mathcal{SCRIPT}$ - Configuração Inicial.

O projeto que estamos propondo consiste no 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 uma tradução direta do $\mathcal{LAMB}$ para $\mathcal{HASKELL}$, permitindo com isso, uma outra alternativa na execução dos programas gerados pelo *Front-End* do compilador de $\mathcal{SCRIPT}$. A utilização de $\mathcal{HASKELL}$ pode ser justificada pelo fato desta linguagem, além de possuir compiladores eficientes, ser uma linguagem de programação avançada, utilizada em aplicações do mundo real e que contém várias características importantes, suportanto inclusive o estilo orientado por objetos.

Em suma, estaremos trabalhando na parte do *Back-End* do compilador, gerando código $\mathcal{HASKELL}$ aproveitando todas as demais partes já implementadas, como a verificação de tipos e a tabela de símbolos [9]. O compilador de $\mathcal{LAMB}$ para $\mathcal{HASKELL}$ e o compilador $\mathcal{HASKELL}$

[1] do inglês, *dynamic binding*

substituirão as etapas de $\lambda-$*lifting* e do *Back-End*. A Figura 2 apresenta o novo esquema proposto para a compilação de programas escritos em $\mathcal{SCRIPT}$.

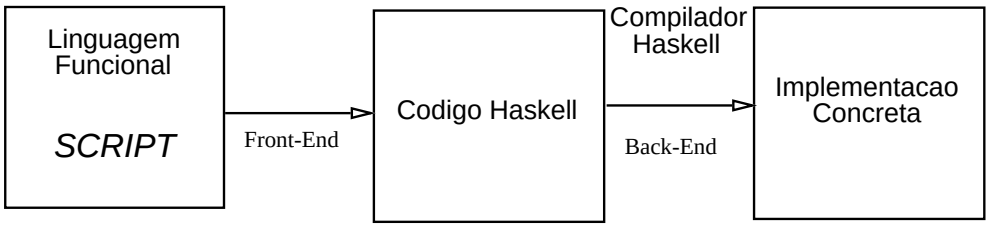

Figura 2: Compilador $\mathcal{SCRIPT}$ - Configuração Alternativa.

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

O compilador será implementado na linguagem C e gerará código $\mathcal{HASKELL}$ na versão 1.3.

3 Revisão da Literatura

3.1 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 multiplicação, 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 distingüem as duas expressões de **max** é denominada *guard*.

As próximas seções apresentam alguns conceitos importantes em linguagens de programação funcional.

3.1.1 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)`.

3.1.2 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[2] 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].

[2] do inglês, *curried*

3.1.3 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.

```
?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 `imprimeA` 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.

Em casos em que a função é capaz de retornar mesmo para um argumento indefinido, diz-se que a função é **não estrita** no argumento. Se uma função é **estrita** em um argumento, então o resultado é indefindo quando um valor indefinido for passado como argumento, e diz-se nestas situações que o argumento tem **avaliação estrita**.

3.1.4 Padrões

Pattern matching é um conceito comum em linguagens funcionais, 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[3] 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 3.2.4) no padrão é **uma ocorrência associada** (veja Seção 3.2.4). 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].

3.2 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.

3.2.1 Notação

Considere novamente a função **dobro** definida na Seção 3.1:

```
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.

3.2.2 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:

[3] do inglês, *pattern*

```
<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 `((`λ `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 (`$\lambda$ 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 3.2.1, 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 (`$\lambda$ `x.(`$E_1\ E_2\ E_3$`))` e não `((`$\lambda$ `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 (`$\lambda$ `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 substituída pela sua definição, antes da redução da expressão lambda. As técnicas de redução de expressões lambda são apresentadas na Seção 3.2.6.

3.2.3 Funções Currificadas

Na Seção 3.1.2, 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) $\rightarrow$ (N $\rightarrow$ N) (d)
   onde **adição** a b = a + b

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

As operações de *currying* e *uncurrying* de 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

3.2.4 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.

Diz-se 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**. 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 $(\lambda\ x.(mul\ x\ y\ ))\ [y \rightarrow x]$ nos leva a $(\lambda\ x.(mul\ x\ x\ ))$. Repare que o significado da expressão original era realizar a multiplicação de dois números, enquanto o objetivo da segunda expressão é calcular o quadrado de um número. Isto ocorre porque a substituição efetuada envolve uma **variável capturada**, ou seja, a variável livre x em [y → 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[4] e E_{rand}, ao operando ou argumento da combinação[5].

[4] do inglês, *operator*
[5] do inglês, *operand*

3.2.5 Tipos de Redução Lambda

A seguir, são apresentados quatro tipos de redução lambda. Cada um deles é utilizado em uma etapa durante o processo de simplificação ou avaliação de uma expressão lambda.

1. **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`

2. **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 denotada 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`.

3. **REDUÇÃO-η:** sejam `v` uma variável e `E` uma expressão lambda. Denota-se a redução-η por:

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

4. **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`

3.2.6 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, diz-se que uma expressão lambda está na **forma normal** se ela não contém *β-redexes*[6], 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)`, é dito que a *β-redex* `E` é **externa** a qualquer *β-redex* que ocorra em `A` ou `B` e que estas são **internas** 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.

A seguir é mostrada 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 seja 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 nenhuma *redex* em uma expressão lambda que esteja na abstração seja 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.

3.3 A Linguagem $\mathcal{LAMB}$

Por razões de completeza, será apresentada a linguagem $\mathcal{LAMB}$ para a qual, inicialmente, foi proposto desenvolver um interpretador.

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

$\mathcal{LAMB}$ 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, $\mathcal{LAMB}$ é usada como linguagem objeto para as linguagens funcionais [3]. A versão de $\mathcal{LAMB}$ aqui apresentada é a mesma utilizada por Oliveira [9] como linguagem intermediária na compilação de programas escritos em $\mathcal{SCRIPT}$.

3.3.1 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 àquela descrita na Seção 3.2.6.

A seguir, apresenta-se 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>)`.

3.3.2 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 pela 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` é possível 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` é possível formar as seguintes expressões $\mathcal{LAMB}$ cujos valores são `TT` se n_1 e n_2 satisfizerem as relações correspondentes, ou `FF` caso contrário:
  - 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` é possível formar as seguintes expressões $\mathcal{LAMB}$ cujos valores são `TT` se t_1 e t_2 satisfizerem as relações correspondentes, ou `FF` caso contrário:
  - 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. Estão disponíveis os 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.

- `e1 IS p`: testa a estrutura, ou seja, a forma, da expressão `e1` 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` é possível 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`, ..., `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. Existem disponíveis os seguintes operadores para construir e manipular árvores sintáticas em $\mathcal{LAMB}$.

  - `[`D_1`, ..., `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`.

3.3.3 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 3.1 pode ser escrita em $\mathcal{LAMB}$ como:

`LAM x.(2` $\times$ `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 $\mathcal{FIXLAM}$ `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))).
```

3.3.4 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 relacionados aos padrões são responsáveis pela ligação destes com as subexpressões correspondentes, de forma que o ambiente necessário para a avaliação da expressão final e está assegurado por esses identificadores.
- `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 $\mathcal{LAMB}$. As expressões que definem ambientes utilizando `DEF` tem função parecida com as expressões que definem ambientes utilizando `LET`. A diferença está no momento da avaliação da expressão final e.

Para expressões com `DEF` cada definição deve ser avaliada no ambiente composto pelas definições com `DEF` que são ligadas entre si (possuem dependência, recursão). As definições com `LET` são avaliadas no ambiente global.

3.3.5 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`, $\texttt{e}_1$, $\texttt{e}_2$, ..., $\texttt{e}_m$ como expressões $\mathcal{LAMB}$, `id` como um identificador e n como uma variável inteira contida em $\mathcal{N}$, pode-se construir um novo padrão das segintes formas:

- $\mathcal{LAM}$ `?.?`: casa com abstrações de funções.
- `(`$\texttt{e}_1$`, ..., `$\texttt{e}_m$`)`: casa com tuplas com m componentes.
- `<`$\texttt{e}_1$`, ..., `$\texttt{e}_m$`>`: casa com listas com m componentes.
- `[`$\texttt{e}_1$`, ..., `$\texttt{e}_m$`]`: 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.
- $\texttt{e}_1$ `PRE` $\texttt{e}_2$`*`: casa com listas com pelo menos um componente.
- $\texttt{e}_1$ `AUG` $\texttt{e}_2$`*`: casa com listas com pelo menos um componente.
- `id[`n`]` `EXT e` ou $\texttt{e}_1$ `EXT` $\texttt{e}_2$: casa com tuplas de tamanho qualquer.
- $\texttt{e}_1$ `NODE` $\texttt{e}_2$: casa com nodos de árvores sintáticas.
- `NUMBER e`: casa com números.
- `TRUTH e`: casa com valores booleanos.
- `QUOTE e`: casa com *quotations*.

3.3.6 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*[7]. $\mathcal{LAMB}$ utiliza especificamente, um tipo de avaliação *lazy* denominada *chamada por necessidade*[8].

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.

3.4 A Linguagem $\mathcal{SCRIPT}$

Por razões de completeza, será apresentada a linguagem $\mathcal{SCRIPT}$, para a qual geraremos código $\mathcal{HASKELL}$.

A linguagem $\mathcal{SCRIPT}$ foi desenvolvida pelo professor 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].

3.4.1 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 `?`.

[7] do inglês, *call by lazy*
[8] do inglês, *call by need*

- **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*[9] 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 `(`$\mathtt{a}_1$`:`$\mathtt{d}_1$`, `$\mathtt{a}_2$`:`$\mathtt{d}_2$`, ..., `$\mathtt{a}_n$`:`$\mathtt{d}_n$`)` onde $\mathtt{d}_i$ é o domínio do elemento $\mathtt{a}_i$, 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. `<`$\mathtt{a}_1$`, `$\mathtt{a}_2$`, ..., `$\mathtt{a}_n$`>`: representa uma instância de uma lista com os componentes $\mathtt{a}_1$, $\mathtt{a}_2$, ..., $\mathtt{a}_n$, todos de um mesmo domínio d.

- **Domínio de funções contínuas:** o domínio das funções contínuas de $\mathtt{d}_1$ a $\mathtt{d}_2$ é denotado pela expressão $\mathtt{d}_1 \rightarrow \mathtt{d}_2$, representando o mapeamento de um domínio $\mathtt{d}_1$ em um domínio $\mathtt{d}_2$.

- **Domínio de nodos:** os nodos de árvores sintáticas com o mesmo rótulo são representados por `[`$\mathtt{D}_1$`, ..., `$\mathtt{D}_n$`]`. O rótulo, que serve para distinguir os nodos, é definido implicitamente pela *quotation* `QUOTE <`$\mathtt{q}_1$`, `$\mathtt{q}_2$`, ..., `$\mathtt{q}_n$`>`, onde $m \leq n$. Cada um dos $\mathtt{q}_i$, $1 \leq i \leq n$, é o nome dos domínios ocorrendo na mesma ordem que em `[`$\mathtt{D}_1$`, ..., `$\mathtt{D}_n$`]`.

3.4.2 Expressões

As expressões em $\mathcal{SCRIPT}$ 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 $\mathcal{SCRIPT}$.

  - **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`;

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

- **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 [$\mathtt{e}_1$, ..., $\mathtt{e}_n$], onde $\mathtt{e}_i$, $1 \leq i \leq n$, é qualquer tipo de expressão.

- **Expressões de padrões:** uma expressão de padrão em $\mathcal{SCRIPT}$ 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, $\mathtt{e}_1$, ..., $\mathtt{e}_n$ expressões de padrões, é possível construir novos padrões como se segue:

  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. $\mathtt{e}_1$ `PRE` $\mathtt{e}_2$`*`: casa com listas com pelo menos um componente.
  5. [$\mathtt{e}_1$, ..., $\mathtt{e}_n$]: 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 $\mathtt{e}_1$ e $\mathtt{e}_2$ expressões em $\mathcal{SCRIPT}$ e p uma expressão de padrão. Existem três formas distintas de comparar expressões em $\mathcal{SCRIPT}$:
  - $\mathtt{e}_1$ `EQ` $\mathtt{e}_2$: testa se as expressões $\mathtt{e}_1$ e $\mathtt{e}_2$ denotam o mesmo valor. É avaliada como `TT` se $\mathtt{e}_1$ e $\mathtt{e}_2$ possuírem o mesmo valor e como `FF` se possuírem valores diferentes ou se pelo menos uma delas possuírem valor funcional.
  - $\mathtt{e}_1$ `NE` $\mathtt{e}_2$: representa a negação da expressão $\mathtt{e}_1$ `EQ` $\mathtt{e}_2$.
  - $\mathtt{e}_1$ `IS` p: testa se a expressão $\mathtt{e}_1$ tem a forma particular, ou seja, a estrutura do padrão p.

- **Expressões condicionais:** uma expressão condicional em $\mathcal{SCRIPT}$ é apresentada como `t` $\rightarrow$ $\mathtt{e}_1$, $\mathtt{e}_2$, correspondendo ao caso onde `t` é uma expressão cuja avaliação retorna `TT`, `FF`, `?` ou $\perp$ e $\mathtt{e}_1$ e $\mathtt{e}_2$ são expressões quaisquer. Caso `t` seja avaliado como `TT`, $\mathtt{e}_1$ será a expressão correspondente; se `t` for avaliado como `FF`, $\mathtt{e}_2$ 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 uma expressão associada ao primeiro padrão que corresponde à estrutura do valor dado. Portanto, para quaisquer expressões `e`, $\texttt{e}_1$, ..., $\texttt{e}_n$ e para os padrões $\texttt{p}_1$, ..., $\texttt{p}_n$, a expressão **CASE** possui a seguinte forma:

$$\begin{array}{l}\texttt{CASE e}\\ \quad \texttt{p}_1 \rightarrow \texttt{e}_1\\ \quad \ldots\\ \quad \texttt{p}_n \rightarrow \texttt{e}_n\\ \quad \texttt{END}\end{array}$$

- **Expressões de abstrações:** são as seguintes expressões de abstrações de $\mathcal{SCRIPT}$:
  - **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:

$$\texttt{LET a}_1 = \texttt{e}_1 \ \texttt{LET a}_2 = \texttt{e}_2 \ \ldots \ \texttt{LET a}_i = \texttt{e}_i \ \texttt{IN e}$$

  onde tem-se $\texttt{a}_i$, $1 \leq i \leq n$, definida no escopo das expressões $\texttt{e}_i$ e e. Cada $\texttt{a}_i$ é dito estar na forma de **ligação de padrões** ou **definição de função**.

- **Aplicações de funções:** em $\mathcal{SCRIPT}$, 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 formas possíveis, que eliminam o uso de parênteses, são `f; g; e` e `f$g(e)`.

Em relação à passagem de parâmetros, $\mathcal{SCRIPT}$ utiliza o mecanismo de avaliação *lazy* de maneira similar ao descrito na Seção 3.1.3.

Outras características importantes de $\mathcal{SCRIPT}$ são: a linguagem permite a definição de funções associadas a domínios de tuplas e sobrecarga[10] de funções.

3.4.3 Estrutura de módulos em $\mathcal{SCRIPT}$

A estrutura completa de uma definição formal em $\mathcal{SCRIPT}$ consiste de um módulo principal, contendo a função principal definida para modelar o contexto geral da definição de semântica denotacional. Além disso, a estrutura pode conter zero ou mais módulos secundários externos que podem ser compilados junto com o módulo principal ou extraído de uma biblioteca de módulos compilados.

A função básica de um módulo é permitir que entidades relacionadas, tais como domínios e funções, sejam agrupadas para poderem ser usadas por outros módulos. Com isso, certos

[10] do inglês, *overloading*

detalhes de definições de módulos e domínios que não precisam ser conhecidos pelos usuários de um módulo podem ficar ocultos.

Há três tipos de módulos em *SCRIPT*: PROJECT, SYNTAX e MODULE.

1. Módulo PROJECT:

   O módulo PROJECT serve para definir os parâmetros e o ambiente sobre o qual as definições devem ser avaliadas.

   Na seção de importação deste módulo, somente uma função pode ser importada e a mesma deve ser considerada como função principal da definição formal.

   Recomenda-se, por razões de clareza, que o domínio da função principal seja redefinido na seção DOMAINS do módulo PROJECT. É necessário que a função principal tenha o mesmo domínio no módulo que a exportou.

   As associações entre arquivos e domínios dos argumentos da função principal são definidas nas seções INFILES e OUTFILES. Tais associações servem para estabelecer onde os argumentos de entrada se encontram e onde os resultados devem ser registrados.

   A seção COMPONENTS apresenta os arquivos dos módulos com as definições formais.

2. Módulo SYNTAX:

   O módulo SYNTAX normalmente apresenta três seções:

   DOMAINS: especifica os domínios de símbolos não-terminais.

   LEXIS: declara as estruturas dos símbolos léxicos.

   SYNTAX: define as sintaxes concretas e abstratas.

3. Módulo MODULE:

   Um módulo MODULE é composto de quatro seções:

   seção EXPORTS: apresenta as entidades do módulo corrente que estão sendo exportadas, assim como seus níveis de encapsulamento.

   seção IMPORTS: são feitas as importações de símbolos, definindo seus graus de visibilidade e relacionando os módulos de onde eles serão importados.

   seção DOMAINS: apresenta as declarações de domínios, variáveis e funções.

   seção DEFINITIONS: apresenta as definições de funções e outros valores, por meio de uma lista de definições de funções.

   Os módulos MODULE servem para fazer encapsulações de definições de domínios e funções, e estabelecer a interface de comunicação entre os módulos.

3.5 A Linguagem *HASKELL*

Esta seção introduz as idéias fundamentais de programação funcional em *HASKELL*. O texto apresentado não cobre toda a linguagem, mas somente as características essenciais para este trabalho. A versão de *HASKELL* apresentada é a 1.3 e maiores detalhes podem ser obtidos em Thompson [14].

3.5.1 Estruturas Básicas da Linguagem

Um programa $\mathcal{HASKELL}$ é uma coleção de *scripts* contendo uma ou mais definição de tipos, variáveis, construtores, módulos e classes.

A seguir, apresenta-se as características básicas da linguagem, extraídas de Thompson [14].

3.5.1.1 Sistema de Tipos: $\mathcal{HASKELL}$ é uma linguagem fortemente tipada e seus tipos podem ser polimórficos, ou seja, podem conter variáveis de tipos que assumem valores de todos os tipos.

Os tipos de dados pré-definidos são:

- **Integer:** conjunto infinito de inteiros ou { `...`, `-2`, `-1`, `0`, `1`, `2`, `...`}.
- **Int:** conjunto finito de inteiros ou {`-max`, `...`, `-2`, `-1`, `0`, `1` ,`2`, `...`, `max`}.
- **Float e Double:** conjunto de números de ponto flutuante.
- **Bool:** contém os valores `"True"` e `"False"`.
- **Char:** contém caracteres como letras, dígitos e caracteres especiais representados entre aspas simples.
- **String:** contém seqüência de caracteres representados entre aspas duplas.

Os usuários podem definir novos tipos em $\mathcal{HASKELL}$. A forma geral de definição de tipos de dados é:

`data` *Typename* `=`
`Con`$_1$ `t`$_{11}$ ... `t`$_{1k_1}$
`Con`$_2$ `t`$_{21}$ ... `t`$_{2k_2}$
...
`Con`$_3$ `t`$_{31}$ ... `t`$_{3k_3}$

Cada `Con`$_i$ é um construtor, seguido por tipos `k`$_i$, onde `k`$_i$ é um inteiro não-negativo.

São possíveis duas extensões de definições:

1. tipos podem ser recursivos; é possível usar o tipo que está sendo definindo, *Typename*, como parte de qualquer um dos tipos `t`$_{ij}$.
2. o *Typename* pode ser seguido por um ou mais variáveis de tipos que podem ser usados no lado direito, tornando a definição polimórfica.

3.5.1.2 Funções: definições de funções consistem de um número de equações condicionais ou não, cada uma podendo ter múltiplas cláusulas. As definições são das seguintes formas:

```
fun p1 p2 ... pn
|g1              = e1
|g2              = e2
|...
|otherwise       = er
```

ou

```
f q1 q2 ... qk
|h1              = f1
|...
|hs              = fs
|...
```

Aqui p_1, q_1, ... são padrões (descritos a seguir); g_1, h_1, ... são expressões booleanas chamadas *guards* e e_1, f_1, ... são expressões que fornecem os resultados da função nos diferentes casos.

Um exemplo de definição é dada pela função que calcula o máximo divisor comum (mdc) entre dois inteiros positivos.

```
mdc ::  Int→Int→Int


mdc n m
|m == n     = n
|m > n       = mdc m n
|otherwise   = mdc (n − m) m
```

3.5.1.3 Estruturas de dados - Tuplas: tuplas são tipos compostos construídos a partir de tipos mais simples. O tipo (`t1, t2, ... tn`), n ≥ 2, consiste de tuplas de valores (`v1, v2, ..., vn`) onde v_i é do tipo t_1, ou equivalentemente, $v_i :: t_i$.

Funções que recebem tuplas como argumentos são definidas por *pattern matching*. Ao invés de se escrever uma variável para um argumento do tipo `(Int, Int)`, um padrão, por exemplo `(x, y)`, é usado.

```
somapar ::  (Int, Int) → Int
somapar (x, y) = x + y
```

Na aplicação, os componentes do padrão são associados aos correspondentes componentes do argumento. Por exemplo:

```
par ::  (Int, Int)
par = (12, 13)

somapar par
= somapar (12, 13)
= 12 + 13
= 25
```

Padrões podem conter constantes e padrões aninhados:

```
inverte ::  ((Int, Int), Int) → ((Int, Int), Int)
inverte ((x, y), z) = ((x, z), y)
```

Funções que acessam partes particulares de uma tupla podem ser definidas por *pattern matching*. Por exemplo, para o tipo `ponto` definido a seguir, as definições poderiam ser:

```
type Ponto = (Float, Float)

coordenada_inicial ::  Ponto → Float
coordenada_final ::  Ponto → Float

coordenada_inicial (n, m) = n
coordenada_final (n, m) = m
```

3.5.1.4 Estruturas de dados - Listas: para qualquer tipo `t`, existe um tipo de `listas de itens` do tipo `t` e seu tipo é escrito `[t]`. Por exemplo,

```
[1, 2, 3, 4, 5, 2, 1] ::  [Int]
[True] ::  [Bool]
```

É possível construir listas de itens de qualquer tipo particular, como por exemplo, listas de funções e listas de listas de números.

```
[[2, 6], [3, 7], []] ::  [[Int]]
```

Como pode ser visto, a lista com os elementos e_1, e_2, ..., e_n é escrita colocando os elementos entre colchetes, ou seja, `[`e_1`, `e_2`, ..., `e_n`]`.

Há um tipo caso especial de lista vazia, `[]`, que não contém itens e é um elemento de todos os tipos de listas.

$\mathcal{HASKELL}$ provê uma forma alternativa de construir listas, chamada de **compreensão de lista**, particularmente útil para construir listas a partir de outras listas bases.

Por exemplo, se a lista `listabase` é `[3, 5, 7]` então

```
[3 * a | a ← listabase]
```

será

```
[9, 15, 21]
```

contendo, deste modo, o triplo de cada elemento `a` da lista `listabase`.

Na compreensão de lista, `a ← listabase` é chamada de **gerador**, porque ele gera os dados a partir dos quais os resultados serão calculados.

3.5.1.5 Definições Locais: cada equação pode ser seguida por uma lista de definições que são locais à função ou outro objeto sendo definido. Estas definições são escritas após a palavra-chave `where`.

Por exemplo, considere a definição da função que retorna a soma do dobro de dois inteiros.

```
somdob x y
= dobx + doby
  where
  dobx = 2 * x
  doby = 2 * y
```

A cláusula `where` deve ser encontrada na definição à qual ela pertence, de tal forma que `where` deve ocorrer em algum ponto à direita do início da definição.

3.5.1.6 Expressões Let: também é possível fazer definições locais para uma expressão. Por exemplo, pode-se escrever

```
let dois = 5 − 3 in dois ∗ 3 + 3 + dois
```

dando o resultado `11`.

3.5.2 Padrões

Padrões em $\mathcal{HASKELL}$ são dados por:

1. constantes inteiras, caracteres e booleanas: `10`, `"b"`, `False`;
2. variáveis: `y`, `variavelsemnome`;
3. tuplas de padrões: por exemplo, $(p_1, p_2, \ldots, p_n)$ onde $p_1, \ldots$ são padrões;
4. listas: `[]` e $(p_1 : p_2)$, onde p_1 e p_2 são padrões;
5. *wild cards*: _;
6. construtores.

Como pode ser visto pelas definições acima, padrões podem ser aninhados: uma tupla ou lista de padrões pode ser construída a partir de outros padrões. Existe outra condição sobre padrões: nenhuma variável pode estar repetida em padrões de uma equação condicional.

Um padrão é utilizado de duas formas: primeiro, para verificar se um argumento possui a forma correta e segundo, para associar valores com variáveis dentro do padrão, de tal modo que valores possam ser usados em uma definição. Por exemplo, se o padrão é utilizado na definição

```
multlista (a:x) = a ∗ multlista x
```

tem-se

```
multlista [1,2,3] = 1 ∗ multlista [2,3].
```

A seguir, são apresentadas as condições nas quais um argumento `a` *casa-se* com um padrão `p`:

- Se `p` for uma constante, então irá *casar* quando `a` for igual a `p`.
- Se `p` for uma variável, por exemplo `x`, `a` *casará* com `p` e será associado a `x` na definição.
- Se `p` for uma tupla de padrões, por exemplo $(p_1, p_2, \ldots, p_n)$, `a` irá *casar* se `a` também for uma tupla, $(a_1, a_2, \ldots, a_n)$, e cada a_i *casar* com p_i.
- Se `p` for um lista de pdrões $(p_1 : p_2)$, `a` *casará* se ele for uma lista não vazia. Sua cabeça é associada a p_1 e sua cauda com p_2.
- Se `p` for um *wild card*, _, então `a` *casará* com `p`, mas nenhuma associação é feita; o *wild card* age simplesmente como um *teste*.

3.5.3 Estrutura de módulos em $\mathcal{HASKELL}$

Um módulo em $\mathcal{HASKELL}$ consiste de um número de definições além de uma interface que define que módulos exportam para outros módulos que usam ou importam tais módulos. Na definição de $\mathcal{HASKELL}$, não há relação entre módulos e arquivos, sendo possível a um arquivo conter vários módulos. Mas como boa prática de programação, recomenda-se escrever um módulo por arquivo.

A seguir, serão apresentados os detalhes de módulos em $\mathcal{HASKELL}$.

- **Módulos cabeçalhos:**

  A cada módulo é dado um nome. Por exemplo, um módulo chamado Formiga pode ser

  ```
  module Formiga where

  data Formigas = ...
  tamandua x = ...
  ```

  Todas as definições devem aparecer nas linhas abaixo da palavra-chave `module` e como regra de segurança, na coluna mais à esquerda do arquivo.

- **Importação de módulos:**

  A operação básica em um módulo é importar um outro módulo. Então, é possível definir o módulo `Abelha` como:

  ```
  module Abelha where

  import Formiga

  apicultor = ... tamandua ....
  ```

  Isto significa que as definições **visíveis** de `Formiga` podem ser usadas em `Abelha`. Por padrão, as definições visíveis em um módulo são aquelas que aparecem no próprio módulo. Se for definido:

  ```
  module Vaca where

  import Abelha
  ```

  as definições de `Formigas` e `Tamandua` não serão visíveis em `Vaca`. Eles podem ser feitos visíveis pela importação explícita de `Formiga`, ou por meio de controles de exportação discutidos a seguir.

- **O módulo principal:**

  Cada sistema de módulos deve conter um módulo de nível superior chamado `main`, que fornece uma definição para o nome `main`. Em um sistema compilado, esta é a expressão que será avaliada quando o código compilado for executado. Em sistemas interpretados, ela possui menor importância.

- **Controles de exportação:**

  Assim como para a cláusula `import`, o padrão é que todas as definições de nível superior de um módulo sejam exportadas.

– É possível não desejar exportar algumas funções auxiliares, tal como a função `shunt` definida a seguir, uma vez que sua única utilidade é na definição da função `reverse`.

```
reverse ::  [t] → [t]
reverse = shunt []

shunt ::  [t] → [t] → [t]
shunt y [] = y
shunt y (a:x) = shunt (a:y) x
```

– É possível desejar exportar algumas das definições que foram importadas de outros módulos. Os módulos `Formiga`, `Abelha` e `Vaca` acima fornecem um exemplo disto.

É possível controlar o que é exportado seguindo o nome de um módulo com uma lista das entidades que devem ser exportadas, Por exemplo, no caso de `Abelha`

```
module Abelha ( apicultor, Formigas (..), tamandua ) whrere ...
```

A lista contém nomes de objetos definidos, tal como `apicultor` e também tipos como `Formigas`. No último caso, segue-se o nome do tipo com `(..)` para indicar que os construtores do tipo são exportados com o próprio tipo.

Tal lista funciona na base de definição-por-definição; é possível afirmar que todas as definições em um módulo serão exportadas, como em

```
module Abelha ( apicultor, module Formiga ) where ...
```

ou de modo similar

```
module Abelha ( module Abelha, module Formiga ) where ...
```

onde a palavra-chave `module` seguida por um nome de módulo tal como `Formiga` é um atalho para todos os nomes definidos no módulo.

- **Controles de importação:**

  É possível controlar de que modo objetos serão importados seguindo a cláusula `import` com uma lista de objetos, tipos ou classes. Por exemplo, ao optar por não importar `tamandua` de `Formiga` escreve-se

  ```
  import Formiga ( Formigas (..)  )
  ```

  afirmando que é desejado apenas o tipo `Formigas`; é possível, alternativamente, explicitar os nomes que se quer *esconder*:

  ```
  import Formiga hiding ( tamandua )
  ```

4 Especificação da Geração para Código $\mathcal{HASKELL}$

4.1 Considerações Iniciais

A seguir, descrevem-se as regras para geração de código $\mathcal{HASKELL}$ utilizadas no terceiro passo da compilação que trata da verificação de tipos e da geração de código final. Primeiramente, serão apresentados os esquemas de tradução que correspondem ao mapeamento das construções de $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$. Em seguida, o esquema de tradução será mostrado em mais baixo nível, por meio de rotinas semânticas associadas às produções da gramática de $\mathcal{SCRIPT}$.

4.2 Convenções

Para os esquemas de tradução que serão apresentados, deve-se considerar que:

- m_1, m_2, ..., m_n são módulos em $\mathcal{SCRIPT}$ do tipo *MODULE*;
- e, e_1, e_2, ..., e_n são expressões $\mathcal{SCRIPT}$;
- p, p_1, p_2, ..., p_n são padrões $\mathcal{SCRIPT}$;
- $f\ (f_1,\ f_2,\ \ldots,\ f_n)$, g e h são identificadores $\mathcal{SCRIPT}$ representando funções;
- $F\ (F_1,\ F_2,\ \ldots,\ F_n)$, G e H representam o corpo de funções em $\mathcal{SCRIPT}$;
- $a\ (a_1,\ a_2,\ \ldots,\ a_n)$, b e c são identificadores $\mathcal{SCRIPT}$ representando variáveis quaisquer ou constantes literais;
- D representa um domínio válido em $\mathcal{SCRIPT}$.

Outras considerações importantes são:

- a compilação de um módulo m é representada por m';
- a compilação de uma expressão e é representada por e$'$;
- a compilação de um corpo de função F é representada por F'; seguindo assim a regra para representar as compilações dos elementos da linguagem.

4.3 Esquemas de Tradução

Nessa seção, apresenta-se o modelo de tradução da linguagem $\mathcal{SCRIPT}$ para a linguagem $\mathcal{HASKELL}$. A abordagem utilizada no processo de geração de código seguirá o esquema de tradução apresentado por Aho [12].

4.3.1 Expressões

1. Expressões Básicas

   (a) Constantes:

SCRIPT	*HASKELL*
number	*number*
quotation	*quotation*
TT	True
FF	False
?	_
THIS	this_

(b) Variáveis:

A tradução de identificadores a em *SCRIPT* para *HASKELL* é a' onde:

- toda ocorrência de $-$ em a é substituída por _1 em a';
- toda ocorrência de $*$ em a é substituída por _2 em a';
- toda ocorrência de $+$ em a é substituída por _3 em a'.

(c) Inteiras:

SCRIPT	*HASKELL*
e_1 PLUS e_2	$e_1' + e_2'$
e_1 MINUS e_2	$e_1' - e_2'$
e_1 MULT e_2	$e_1' * e_2'$
e_1 DIV e_2	e_1' `` `div` `` e_2'
e_1 REM e_2	e_1' `` `mod` `` e_2'
NEG e	negate e'
e_1 EQ e_2	$e_1' == e_2'$
e_1 NE e_2	$e_1' /= e_2'$
e_1 LT e_2	$e_1' < e_2'$
e_1 GT e_2	$e_1' > e_2'$
e_1 LE e_2	$e_1' <= e_2'$
e_1 GE e_2	$e_1' >= e_2'$

(d) *Quotation*

SCRIPT	*HASKELL*
e_1 EQ e_2	$e_1' == e_2'$
e_1 NE e_2	$e_1' /= e_2'$
e_1 LT e_2	$e_1' < e_2'$
e_1 GT e_2	$e_1' > e_2'$
e_1 LE e_2	$e_1' <= e_2'$
e_1 GE e_2	$e_1' >= e_2'$
e_1 CAT e_2	e_1 ++ e_2
QUOTE e	e'
NUMBER e	read e'::Int
TRUTH e	f_truth e'

(e) Expressões Lógicas:

SCRIPT	*HASKELL*
e_1 AND e_2	$e_1^{'}$ && $e_2^{'}$
e_1 OR e_2	$e_1^{'}$ || $e_2^{'}$
NOT e	not $e^{'}$
e_1 EQ e_2	$e_1^{'}$ == $e_2^{'}$
e_1 NE e_2	$e_1^{'}$ /= $e_2^{'}$

(f) Expressões de Listas:

SCRIPT	*HASKELL*
$<e_1, e_2, \ldots, e_n>$	$[e_1^{'}, e_2^{'}, \ldots, e_n^{'}]$
$<e_1$ .. $e_2>$	$[e_1^{'}$ .. $e_2^{'}]$
$< e \mid q_1 \mid q_2 \mid \ldots \mid q_n>$ onde q_i, $1 \leq i \leq n$ e $n \geq 1$, tem a seguinte forma: • $p_i \leftarrow e_i$	$[e \mid q_1^{'}, q_2^{'}, \ldots, q_n^{'}]$. A compilação dos $q_i^{'}$s é: • $p_i^{'} \leftarrow e_i^{'}$
$< e \mid q_1 \mid q_2 \mid \ldots \mid q_n>$ onde q_i, $1 \leq i \leq n$ e $n \geq 1$, tem a seguinte forma: • $p_i \leftarrow e_{1i}::e_{2i}$	$[e \mid q_1^{'}, q_2^{'}, \ldots, q_n^{'}]$. A compilação dos $q_i^{'}$s é: • $p_i^{'} \leftarrow e_{1i}^{'}, e_{2i}^{'}$
e $\{u_1\}$ $\{u_1\}$ ... $\{u_n\}$ onde u_i tem a forma $e_{1a} = e_{1b}, e_{2a} = e_{2b}, \ldots, e_{ma} = e_{mb}$	list_atua $e^{'}$ $[e_{1a}^{'}, e_{2a}^{'}, \ldots, e_{ma}^{'}]$ $[e_{1b}^{'}, e_{2b}^{'}, \ldots, e_{mb}^{'}]$
e_1 CAT e_2	$e_1^{'}$ ++ $e_2^{'}$
CONC e	concat $e^{'}$
e_1 AUG e	$e_1^{'}$ ++ $[e^{'}]$
e PRE e_1	$e^{'}$: $e_1^{'}$
SIZE e	length $e^{'}$
e_1 EL n	$e_1^{'}$!! n
HEAD e	head $e^{'}$
TAIL e	tail $e^{'}$

(g) Expressões de Tuplas:

SCRIPT	*HASKELL*
$(e_1,\ e_2,\ \ldots,\ e_n)$	$(e_1^{'},\ e_2^{'},\ \ldots,\ e_n^{'})$
e_1 CAT e_2	Considerando que a tupla e_1 contém k elementos e a tupla e_2, j elementos, temos: let $(x_1,\ x_2,\ \ldots,\ x_k) = e_1$ in let $(y_1,\ y_2,\ \ldots,\ y_j) = e_2$ in $(x_1,\ x_2,\ \ldots,\ x_k,\ y_1,\ y_2,\ \ldots,\ y_j)$

(h) Expressões com Nodos:

SCRIPT	*HASKELL*
$[e_1,\ e_2,\ \ldots,\ e_n]$	Considerando que o domínio de a_1 é D_1, o domínio de a_2 é D_2, ..., e o domínio de a_n é D_n, temos: $([D_1,\ D_2,\ \ldots,\ D_n],\ (a_1,\ a_2,\ \ldots,\ a_n))$

2. Expressões de Comparação:

SCRIPT	*HASKELL*
e_1 EQ e_2	$e_1^{'} == e_2^{'}$
e_1 NE e_2	$e_1^{'} /= e_2^{'}$
e_1 IS e_2	case $e_1^{'}$ of $e_2^{'} \rightarrow$ True otherwise $\rightarrow$ False

3. Expressões Condicionais:

SCRIPT	*HASKELL*
$e_1 \rightarrow e_2$ ELSE e_3	if $e_1^{'}$ then $e_2^{'}$ else $e_3^{'}$
$e_1 \rightarrow e_2,\ e_3$	if $e_1^{'}$ then $e_2^{'}$ else $e_3^{'}$

4. Expressões de Abstrações:

SCRIPT	*HASKELL*
LAM $p.e$	$\backslash p^{'} \rightarrow e^{'}$

5. Expressões LET:

SCRIPT	*HASKELL*
LET $\text{lhs}_1 = \text{C}_1$ LET $\text{lhs}_2 = \text{C}_2$ LET $\text{lhs}_3 = \text{C}_3$ $\vdots$ LET $\text{lhs}_{n-1} = \text{C}_{n-1}$ LET $\text{lhs}_n = \text{C}_n$ IN e	let $\text{lhs}^{'}_1 = \text{C}^{'}_1$; $\text{lhs}^{'}_2 = \text{C}^{'}_2$; $\text{lhs}^{'}_3 = \text{C}^{'}_3$; $\vdots$ $\text{lhs}^{'}_{n-1} = \text{C}^{'}_{n-1}$; $\text{lhs}^{'}_n = \text{C}^{'}_n$; in $e^{'}$

6. Expressões CASE:

SCRIPT	*HASKELL*
CASE e $/p_{11}/p_{12}/\ldots/p_{1n} \rightarrow e_1$ $/p_{21}/p_{22}/\ldots/p_{2n} \rightarrow e_2$ $\vdots$ $/p_{j1}/p_{j2}/\ldots/p_{jk} \rightarrow e_j$ END	case $e^{'}$ of $p^{'}_{11} \rightarrow e^{'}_1$ $\vdots$ $p^{'}_{1n} \rightarrow e^{'}_1$ $p^{'}_{21} \rightarrow e^{'}_2$ $\vdots$ $p^{'}_{2m} \rightarrow e^{'}_2$ $\vdots$ $p^{'}_{ji} \rightarrow e^{'}_j$ $\vdots$ $p^{'}_{jk} \rightarrow e^{'}_j$

7. Expressões Seqüenciais:

SCRIPT	*HASKELL*
$e_1\ e_2$	$e^{'}_1\ e^{'}_2$
e_1 ; e_2	$e^{'}_1\ e^{'}_2$
e_1 \$ e_2	$e^{'}_1\ e^{'}_2$

8. Avaliação de Parâmetros:

SCRIPT	*HASKELL*
VAL e	`val_`$_i$ onde `val_`$_i$ = e'

9. Expressões de Entrada e Saída:

SCRIPT	*HASKELL*
OUT e	f_out e'

10. Expressões de Domínios:

- Declaração de função associada a domínio:

SCRIPT	*HASKELL*
DEF $A.f$ (a_1:D_1,a_2:D_2, ..., (a_n:D_n):D_{n+1}	f_A *this* a_1 a_2 ... a_n

- Seleção de campos associados a expressões

SCRIPT	*HASKELL*
$this.a_n$	Considerando m ($m \geq n$) como número de campos do *this*, temos: let (a_1, a_2, ..., a_m) = *this* in a_n
$e.a_n$	Considerando m ($m \geq n$) como número de campos da expressão e, temos: let (a_1, a_2, ..., a_m) = e in a_n

- Chamada de função associada a domínio:

SCRIPT	*HASKELL*
$a.f\ (x_1)\ (x_2)\ \ldots\ (x_n)$	$f\_\_A\ a\ (x_1)\ (x_2)\ \ldots\ (x_n)$ onde A é o domínio da expressão a

- Equivalência entre parâmetros e argumento:

SCRIPT	*HASKELL*
DEF $z = p\ (a)$	Seja m o número de campos associados ao parâmetro ao qual o argumento a está ligado, sendo n o número de campos do argumento a. Temos dois casos: (a) se $m = n$, a compilação é: $z = p\ a$ (b) se $n > m$, a compilação é: $z = p$ (let $(a_1, a_2, \ldots, a_n) = a$ in $(a_1, a_2, \ldots, a_m)$)

- Atribuição entre tuplas:

SCRIPT	*HASKELL*
DEF $d{:}A = c$ onde A é um domínio de tupla.	Seja n o número de campos de d e m o número de campos de c. Temos dois casos: (a) se $m = n$, a compilação é: $d = c$ (b) se $n > m$, a compilação é: $d =$ let $(c_1, c_2, \ldots, c_m) = c$ in $(d_1, d_2, \ldots, d_n)$)

4.3.2 Padrões

SCRIPT	*HASKELL*
p_1 PRE p_2	$p_1^{'}$:$p_2^{'}$
p_1 CAT p_2	• no caso do tipo de p_1 e p_2 ser tupla: Considerando que a tupla p_1 contém k elementos e a tupla p_2, j elementos, temos: let (x_1, x_2, ..., x_k) = p_1 in let (y_1, y_2, ..., y_j) = p_2 in (x_1, x_2, ..., x_k, y_1, y_2, ..., y_j) • no caso do tipo de p_1 e p_2 ser lista: $p_1^{'}$ ++ $p_2^{'}$
NUMBER p	read $p^{'}$::Int
QUOTE p	$p^{'}$
TRUTH p	f_truth $p^{'}$
VAL p	`val_`i onde `val_`i = $p^{'}$
(p_1, p_2, ..., p_n)	($p_1^{'}$, $p_2^{'}$, ..., $p_n^{'}$)
[a_1, a_2, ..., a_n]	Considerando que o domínio de a_1 é D_1, o domínio de a_2 é D_2, ..., e o domínio de a_n é D_n, temos: ([D_1, D_2, ..., D_n], (a_1, a_2, ..., a_n))
a	a$^{'}$
< >	[ ]

4.3.3 Definições

SCRIPT	*HASKELL*
DEF $p = e$	$p' = e'$
DEF $f\ p_1\ p_2\ \ldots\ p_n = e$	$f'\ p_1'\ p_2'\ \ldots\ p_n' = e'$
DEF $f\ p_1\ p_2\ \ldots\ p_n : D = e$	$f'\ p_1'\ p_2'\ \ldots\ p_n' = e'$
DEF $D.f\ p_1\ p_2\ \ldots\ p_n = e$	$D\_f'\ p_1'\ p_2'\ \ldots\ p_n' = e'$
DEF $D.f\ p_1\ p_2\ \ldots\ p_n : D = e$	$D\_f'\ p_1'\ p_2'\ \ldots\ p_n' = e'$

4.3.4 Módulos

- Considerando o módulo m_i como sendo o módulo apresentado a seguir:

MODULE m_i
EXPORT p_1, p_2, p_3, ..., p_m
DEFINITIONS
DEF $f_1 = F_1$
DEF $f_2 = F_2$
DEF $f_3 = F_3$
$\vdots$
DEF $f_{n-1} = F_{n-1}$
DEF $f_n = F_n$
END m_i

A compilação de m_i para $\mathcal{HASKELL}$ é:

module m_i (p'_1,p'_2,p'_3, ..., p'_m) where

$f'_1 = F'_1$
$f'_2 = F'_2$
$f'_3 = F'_3$
$\vdots$
$f'_{n-1} = F'_{n-1}$
$f'_n = F'_n$

- Considerando o módulo m_j como sendo o módulo apresentado a seguir:

MODULE m_j
IMPORT m_i (p_1, p_3, ..., p_x, ..., p_y)
IMPORT m_k(g)
DEFINITIONS
DEF $h = \ldots p_1 \ldots g \ldots p_3 \ldots$
END m_j

onde $1 \leq x, y \leq m$, ou seja, p_x e p_y são quaisquer das funções exportadas por m_i. A compilação do módulo m_j será:

module m_j where

import m'_i (p'_1,p'_3, ..., p'_x, ..., p'_y)
import m'_k (g')

$h = \ldots\ p'_1\ \ldots\ g'\ \ldots\ p'_3$

4.4 Rotinas Semânticas

Nas produções apresentadas a seguir, devem ser consideradas as seguintes convenções:

- não terminais: são palavras do alfabeto latino em letras minúsculas;
- terminais: símbolos em negrito ou seqüências de caracteres entre aspas (“ ”).

É importante ressaltar que somente serão apresentadas nesta seção aquelas produções que demandam rotinas semânticas para gerar código $\mathcal{HASKELL}$. As demais permanecem inalteradas, como apresentadas originalmente por Fabíola Oliveira [9].

Para expressar as rotinas semânticas associadas às produções $\mathcal{SCRIPT}$ será utilizada a seguinte convenção:

- Inicialmente apresenta-se a produção da gramática seguida da definição da ação semântica que trata da geração de código, utilizando-se de uma metalinguagem com características próximas à linguagem C. Se necessário, é descrito em português a semântica do que deverá acontecer.

1. `module ::= "MODULE" module_ide mod_section* "END" module_ide`
   - `gera ("module", ModCorrente, "where")`
2. `definition ::= lhs "=" exp`
   - Análise de `lhs.tipo`
     - Se o valor de `lhs.tipo` fôr _simples então `gera (lhs.tokenlist, "=")`
     - Senão `gera_lista (lhs.tok_walker, "=")`
   - Geração de código referente a `exp` (vide regras 3, 4, 5, 6, 7)
3. `exp ::= "LAM" pattern_exp "." exp`
   - strcpy (`s_mod`, “\”)
   - `gera (s_mod)`
   - Geração de código referente a `pattern_exp`
   - strcpy `s_mod`, “.”)
   - `gera (s_mod)`
   - Geração de código referente a `exp` (vide regras 3, 4, 5, 6, 7)

4. `exp ::= exp_a "→" exp else_symbol exp`

   - strcpy (`s_mod`, "if")
   - `gera (s_mod)`
   - Geração de código referente a `exp_a` cujo conteúdo está armazenado em `reg_aux_ant1`: `gera (reg_aux_ant1)`
   - strcpy (`s_mod`, "then")
   - `gera (s_mod)`
   - Geração de código referente a `exp` cujo conteúdo está armazenado em `reg_aux_ant2`: `gera (reg_aux_ant2)`
   - strcpy (`s_mod`, "else")
   - `gera (s_mod)`
   - Geração de código referente a `exp` cujo conteudo está armazenado em `reg_aux_atu`: `gera (reg_aux_atu)`

5. `exp_a upda_exp_list`

   - `strcpy (s_tok, "atualiza_lista")`
   - `gera (s_tok)`
   - Geração do código referente a `exp_a` cujo conteúdo está armazenado em `_list_origem`: função `Gera_lista_Haskell (_list_origem)` gera o código apropriado
   - Geração do código referente a `upda_exp_list` cujo conteúdo está armazenado em `_list_atualizacao`: função `Gen_Lista_Atua_Haskell (_list_atualizacao)` gera o código apropriado.

6. `exp_f ::= variable`

   - strcpy (`s_ide`, `variable.ElementoCP`)
   - `gera (s_di)`

7. `exp_f ::= literal_const`

   - strcpy (`s_di`, `literal_const.ElementoCP`)
   - `gera (s_di)`

8. `list_exp ::= "<" ">"`

   - strcpy (`s_di1`, "[")
   - strcpy (`s_di2`, "]")
   - `gera (s_di1)`
   - `gera (s_di2)`

9. `list_exp ::= "<" exp_list_com ">"`

- strcpy (s_di1, "[")
- strcpy (s_di2, "]")
- gera (s_di1)
- Geração de código referente a exp_list_com cujo conteúdo está armazenado em tok_aux: Gen_Lista_Virg_Haskell (tok_aux)
- gera (s_di2)

10. list_exp ::= "<" exp ".." exp ">"

    - strcpy (s_mod, "[")
    - Geração de código referente a exp ".." exp cujo conteúdo está armazenado em lista_tokens e updtLisTok[i_updtLisTok]->padrao: função GeraTokListUntil() gera código apropriado.
    - strcpy (s_mod, "]")
    - gera (s_mod)

11. list_exp ::= "<" exp qualif_list ">"

    - strcpy (s_mod, "[")
    - gera (s_mod)
    - Geração de código referente a exp cujo conteúdo está armazenado em list_exp: gera_lista (list_exp)
    - Geração de código referente a qualif_list cujo conteúdo está armazenado em qualif_list.lista: função GeraTokListCompr() gera o código apropriado.
    - strcpy (s_mod, "]")
    - gera (s_mod)

12. tuple_exp ::= "(" ")"

    - strcpy (s_di1, "(")
    - strcpy (s_di2, ")")
    - gera (s_di1)
    - gera (s_di2)

13. tuple_exp ::= "(" exp_com_list ")"

    - strcpy (s_di1, "(")
    - strcpy (s_di2, ")")
    - gera (s_di1)
    - Geração de código referente a exp_list_com cujo conteúdo está armazenado em tok_aux: Gen_Lista_Virg_Haskell_Tupla (tok_aux)
    - gera (s_di2)

14. `node_exp ::= "[" "]"`

    - strcpy (`s_aux`, “(”)
    - `gera (s_aux)`
    - strcpy (`s_aux`, ”?”)
    - `gera (s_aux)`
    - strcpy (`s_aux`, “)”)

15. `node_exp ::= "[" exp_g_list "]"`

    - strcpy (`s_aux`, “(”)
    - `gera (s_aux)`
    - strcpy (`s_aux`, ”?”)
    - `gera (s_aux)`
    - strcpy (`s_aux`, ”, (”)
    - `gera (s_aux)`
    - Geração de código referente a `exp_g_list` cujo conteúdo está armazenado em `listaTokens_aux`: função `Gen_Lista_Virg_Haskell (listaTokens_aux)` gera o código apropriado.
    - strcpy (`s_aux`, “)”)

16. `case_exp ::= "CASE" exp_a clause_list "END"`

    - strcpy (`s_case`, “case”)
    - `gera (s_case)`
    - Geração de código referente a `exp_a` cujo conteúdo está armazenado em `tok_aux`: função `Gen_Lista_Tokens()` gera o código apropriado.
    - strcpy (`s_mod`, “of”)
    - `gera (s_mod)`
    - Geração de código referente a `clause_list` cujo conteúdo foi dividido em três partes e armazenado em:
      - `nome_tok_aux`
      - `list_walker`→`padrão`
      - `list_arrow`

      A função `GeraTokClausLst()` gera o código apropriado.

17. `exp_a ::= exp_b "IS" pattern_exp`

    - strcpy (`s_mod`, “case”)
    - `gera (s_mod)`
    - strcpy (`s_mod`, `exp_b.ElementoCP`)

- `gera (s_mod)`
- strcpy (`s_mod`, "of")
- `gera (s_mod)`
- Geração de código referente a pattern_exp (conteúdo de `pattern_exp` armazenado em `tokenList`): `gera_lista (tokenList)`
- strcpy (`s_mod`, "→ True")
- `gera (s_mod)`
- strcpy (`s_mod`, "otherwise → False")
- `gera (s_mod)`

18. `let_binding ::= "LET" definition`

    - strcpy (`s_mod`, "let")
    - `gera (s_mod)`
    - Geração de código referente a `definition` (vide regra 2)
    - strcpy (`s_mod`, "in")
    - `gera (s_mod)`

19. `field_qualif ::= obj_exp "." obj_exp_list`

    - Geração de código referente a `obj_exp` cujo conteúdo está armazenado em `DT_ParD`
    - `strcpy (s_tok, ".")`
    - `gera (s_tok)`
    - Geração de código referente a `obj_exp_list` cujo conteúdo está armazenado em `list_aux`

5 Implementação do Compilador $\mathcal{SCRIPT}$

5.1 Introdução

Por razões de completeza, a próxima subseção deste capítulo apresenta suscintamente como foi implementado o *front-end* do compilador $\mathcal{SCRIPT}$, baseado na tese de Fabíola Oliveira [9], que apresenta maiores detalhes sobre esta fase. Isto facilitará o entendimento do que foi feito no *back-end* do compilador, mais especificamente na geração de código, exposto na Seção 5.3.

5.2 Visão Geral do Compilador

O projeto do compilador $\mathcal{SCRIPT}$ [2, 9] foi estruturado em três passos, como mostrado na Figura 3, extraída de Fabíola Oliveira [9]. A organização em três fases se fez necessária, uma vez que a aleatoriedade de programas escritos em $\mathcal{SCRIPT}$ pode fazer com que informações necessárias em um determinado momento ainda não estejam disponíveis, principalmente nos casos de rotinas semânticas recursivas e de declarações que aparecem após o uso. A seguir, é apresentada uma descrição de cada um dos passos.

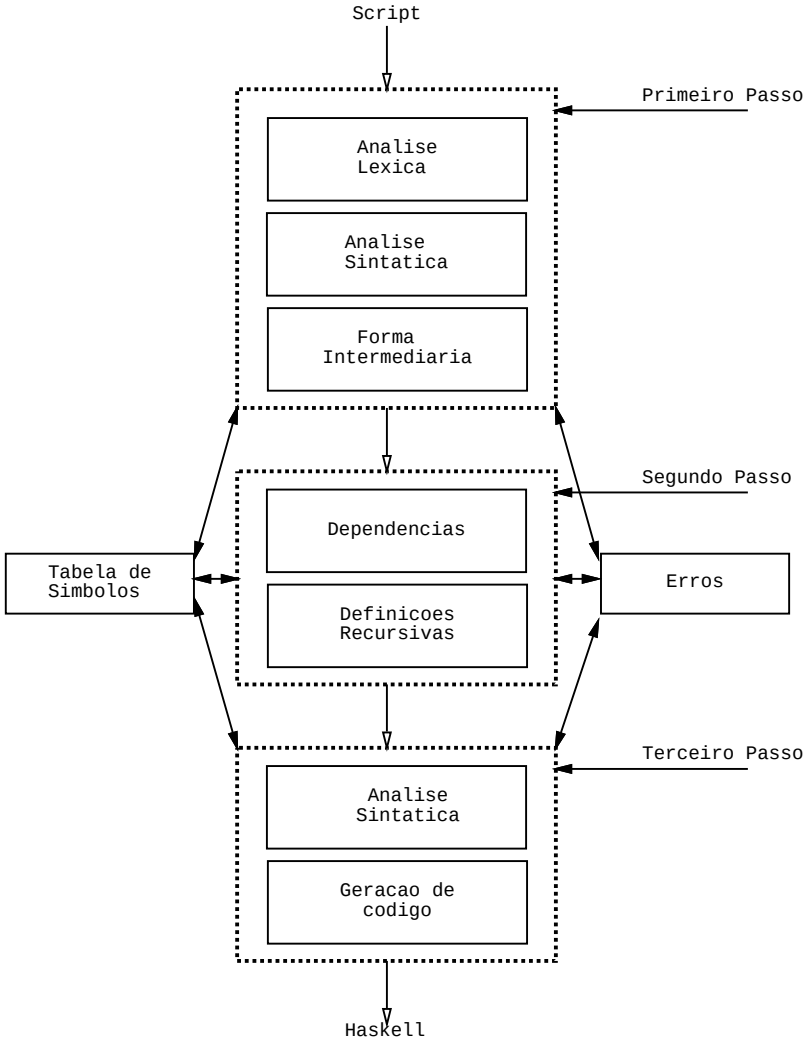

Figura 3: Projeto do Compilador $\mathcal{SCRIPT}$.

Primeiro Passo: As principais tarefas do primeiro passo são:

1. processamento do texto fonte;
2. análise léxica;
3. análise sintática;
4. coleta e instalação na tabela de símbolos de domínios e variáveis.

- A análise léxica foi feita utilizando-se a linguagem **Lex** [16], que após compilação gerou o analisador léxico. Os serviços executados por este módulo são:
  - reconhecimento e agrupamento de *itens léxicos* ou *tokens*;
  - reconhecimento de erros léxicos.
- A análise sintática corresponde à rotina principal do *Front-End* do compilador. Sua tarefa é reconhecer a estrutura do programa de acordo com as regras de formação de unidades sintáticas fornecidas pela gramática. Para tal, o analisador sintático obtém do analisador léxico o fluxo de *itens sintáticos* bem como as informações sobre esses itens. Nesta fase utilizou-se o **Yacc** [17] para gerar o analisador sintático, cuja entrada foi uma gramática que segue a especificação LALR [12].
- A tabela de símbolos executa as tarefas de:
  - gerência de escopo e visibilidade dos símbolos;
  - inserção, remoção e consulta de símbolos;
  - exportação e importação de símbolos.

Segundo Passo: A fase de análise de dependência e recursividade entre funções e variáveis consistiu na criação de listas para indicar quais identificadores aparecem livres em quais definições ou funções. A coleta deste tipo de informações termina com a incorporação das mesmas aos símbolos instalados na tabela de símbolos.

Terceiro Passo: Esta etapa consistiu na verificação de tipos, análise semântica e geração de código.

- A verificação de tipos oferece o serviço de determinação de equivalência e compatibilidade de domínios, bem como detecção de erros relacionados ao uso de domínios.
- A análise semântica verifica a estrutura semântica do programa fonte, relatando erros semânticos estáticos. As informações coletadas sobre as construções serão utilizadas na fase de geração de código. Sendo assim, é uma questão de projeto, associar a cada símbolo não-terminal da gramática um conjunto de atributos e a cada produção da gramática um conjunto de regras semânticas que manipularão os valores dos atributos associados aos símbolos não-terminais da gramática.
- A geração de código $\mathcal{HASKELL}$, assunto principal deste trabalho, será abordada na seção seguinte.

5.3 Implementação do Gerador de Código

5.3.1 Recuperação de Informações da Tabela de Símbolos

Para implementação do compilador de $\mathcal{SCRIPT}$ foi necessário acessar as informações previamente armazenadas na tabela de símbolos. A tabela de símbolos de todo o programa pode ser vista como uma **floresta**, uma vez que para cada módulo a ser compilado, foi montada uma árvore com as informações dos símbolos deste módulo. As estruturas de dados referentes à tabela de símbolos são:

- `SymTableCellT` define um tipo com o objetivo de montar uma árvore da tabela de símbolos para o módulo a ser compilado. Estão contidos neste tipo, as informações de nome do símbolo e de endereços dos nodos da árvore à direita e à esquerda deste símbolo, além dos seus atributos. Para o armazenamento dos atributos foi criado o tipo `AttrTypeT`;

- `AttrTypeT` contém as informações definidas como atributos, incluindo o nível, o tipo do símbolo e a árvore de domínio;

- `DomTreeT` armazena a árvore de domínio de um símbolo, sendo referenciada por um objeto do tipo `AttrTypeT`.

Como a geração de código ocorre somente no terceiro passo da compilação, não foi preciso implementar nenhum procedimento para instalar símbolos ou mesmo atualizar os já instalados. No entanto, se fez necessário conhecer as estruturas de dados bem como os procedimentos implementados para recuperar informações da tabela de símbolos.

As informações recuperadas da tabela de símbolos foram incorporadas aos atributos associados aos símbolos não-terminais da gramática, sendo, portanto, muitas vezes desnecessário acessar a tabela de símbolos, uma vez que tais informações podiam ser obtidas por meio destes atributos. Entretanto, devido a diferenças existentes entre a linguagem $\mathcal{LAMB}$, inicialmente usada como linguagem objeto [4, 9], e a linguagem $\mathcal{HASKELL}$, as informações necessárias para gerar código em cada uma das linguagens nem sempre eram as mesmas. Para o caso de tuplas, a informação do número de componentes não estava disponível pois não era necessária para gerar código $\mathcal{LAMB}$. No entanto, para geração de código $\mathcal{HASKELL}$ esta informação era importante e foram implementados dois procedimentos que recuperavam tal informação. Os módulos implementados são listados a seguir:

```
int GetNumberofFields (DomTreeT *DT_aux_func)

{
  DomTreeT *DT_ParD_aux;
  int numero_campos = 0;

  DT_ParD_aux = DT_aux_func;
  while (DT_ParD_aux)
   {
     numero_campos++;
     DT_ParD_aux = DT_ParD_aux->Brother;
   }
```

```
  return numero_campos;
}

-------------------------------------------------------

int GetNumberofFieldsParD (DomTreeT *DT_aux_func)

{
  DomTreeT *DT_ParD_aux;
  int numero_campos = 0;

  DT_ParD_aux = DT_aux_func->Son;

  while (DT_ParD_aux)
   {
     numero_campos++;
     DT_ParD_aux = DT_ParD_aux->Brother;
   }

  return numero_campos;

}
```

Nos demais casos, as informações foram obtidas por meio de procedimentos já implementados ou diretamente pelos atributos manipulados pelas rotinas semânticas das produções.

5.3.2 Geração de Código

O maior esforço para implementação do gerador de código foi adaptar o mecanismo de geração de código $\mathcal{LAMB}$ para $\mathcal{HASKELL}$. As produções estavam todas estruturadas de modo a gerar código $\mathcal{LAMB}$, dificultando a geração do código $\mathcal{HASKELL}$que não necessariamente possuía a mesma ordem do código $\mathcal{LAMB}$. Por isso, em muitos casos, era preciso armazenar as informações em estruturas temporárias e só ao fim da ação semântica processar o código $\mathcal{HASKELL}$. Por motivos de espaço, o programa escrito na linguagem **Yacc** não será listado.

Foram implementadas ou modificadas as seguintes funções para a geração de código $\mathcal{HASKELL}$:

- `int GeraTokListCompr (RegTokenT *first_list, RegTokenListT *rest_list, int indice)`: utilizada na geração de código de compreensão de listas;
- `int GeraTokLisTupUpdat (RegTokenListT *_list_origem, RegTokenT _list_atualizacao, int _qtos)`: utlizada na geração de código de atualização de listas;
- `int GeraTokDiOp (enum OperadorT op_type, DomTreeT *_domtree)`: utlizada na geração de código de *tokens* de uma expressão com um operador binário;
- `int GeraTokMonOp (enum OperadorT op_type)`: utilizada na geração de código de *tokens* de uma expressão com um operador unário;
- `GeraTokClausLst (char first_list[15], RegTokenT *second_list, RegTokenT third_list)`: utilizada na geração de código de *tokens* de uma lista de cláusulas em uma construção **case**;

- `int GeraTokListUntil ()`: utilizada na geração de código de lista expressas por um intervalo;

- `int Gen_Lista_Haskell (RegTokenT *lista, FILE *fileOut_Tok)`, `int Gen_Lista_Atua_Haskell (RegTokenT *lista, FILE *fileOut_Tok)` e `int Gen_Lista_Virg_Haskell (RegTokenT *lista, FILE *fileOut_Tok)`: utilizada para separar os *itens léxicos* básicos de uma estrutura mais complexa;

- `int Gen_Haskell (int marca, char *token, FILE *fileHaskell)`: utilizada para gravação do código $\mathcal{HASKELL}$ referentes aos *itens léxicos* no arquivo de saída.

5.3.3 Execução do Compilador $\mathcal{SCRIPT}$

A fim de facilitar a execução do compilador $\mathcal{SCRIPT}$, os compiladores referentes aos três passos são invocados a partir de um único arquivo escrito em $\mathcal{PERL}$. O compilador foi desenvolvido para executar em ambientes **UNIX**. A sintaxe de chamada é:

cs.pl [arq_ent] [arq_sai]

onde [arq_ent] indica o arquivo de entrada contendo um programa $\mathcal{SCRIPT}$ e [arq_sai] indica o arquivo que conterá o mesmo programa compilado para $\mathcal{HASKELL}$.

6 Conclusão

O compilador $\mathcal{SCRIPT}$ descrito está inserido em um ambiente cujo objetivo é oferecer aos projetistas de linguagens de programação uma ferramenta para composição de descrições de semântica denotacional de maneira estruturada. A utilização de tal compilador visa o aumento da qualidade dessas descrições, ou seja, definições mais confiáveis e fáceis de ler e compreender.

O trabalho compreendeu a implementação do gerador de código $\mathcal{SCRIPT}$ que fará parte do *back-end* do compilador $\mathcal{SCRIPT}$, em substituição ao gerador de código $\mathcal{LAMB}$, originalmente implementado por Fabíola Oliveira [9]. As etapas do *Front-end* do compilador como análises léxica e sintática, verificação de tipos e tabela de símbolos foram as mesmas implementadas por Fabíola Oliveira [9], o que diminuiu o esforço final para a produção deste compilador.

A utilização de $\mathcal{HASKELL}$, linguagem para a qual já existe um compilador disponível, permitiu a obtenção direta de código executável, sem a necessidade de etapas intermediárias.

Este compilador foi implementado na linguagem **C** [7] para o ambiente **UNIX**. O trabalho de programação foi feito utilizando as ferramentas **Lex** e **Yacc**.

Por fim, gostaria de manifestar minha mais profunda gratidão à professora Mariza Bigonha pela orientação e estímulo, tão necessários para o desenvolvimento deste projeto. Também ao professor Roberto Bigonha pelas importantes sugestões e à Fabíola Oliveira pela disponibilidade e ajuda.

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 $\mathcal{SCRIPT}$ 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. *$\mathcal{LAMB}$ - 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.

[14] Thompson, Simon. *Haskell - The Craft of Functional Programming.* Addison-Wesley, 1996.

[15] Turner, D.A. *A New Implementation Technique for Aplicative Languages. Software - Practice and Experience.* 9:31-49, 1979.

[16] Lesk, M. E. Lesk and Schmidt, E. *Lex: A lexical analyzer generator.* Bell laboratories, Murray Hill, New Jersey, 1978.

[17] Inc. Free software Foundation. Gnu projext e compiler. Relatório Técnico, 1988.

A Apêndice A

A.1 Considerações Iniciais

Esta seção apresenta os resultados obtidos com a execução do compilador $\mathcal{SCRIPT}$ para alguns exemplos de programas escritos em $\mathcal{SCRIPT}$. Os exemplos mostrados não constituem programas completos em $\mathcal{SCRIPT}$, mas procuram cobrir todas as estruturas e construções importantes da linguagem. A ordem de apresentação dos exemplos é a seguinte: primeiro, lista-se o código do programa escrito em $\mathcal{SCRIPT}$ que serviu como entrada para o compilador. A seguir, é listado o código $\mathcal{HASKELL}$ obtido como saída execução do compilador. Nos exemplos 13 e 14, além dos programas fonte e objeto, é mostrada a execução destes programas no interpretador **Hugs**.

A.2 Resultados da Execução do Compilador $\mathcal{SCRIPT}$

1. Exemplo de programa com os operadores binários e unários que operam sobre o domínio de inteiros ($\mathcal{N}$)

```
--------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N;
           Tdom = T

DEFINITIONS

DEF ndom1 =  15
DEF ndom2 =  20
DEF ndom3 = ndom2 PLUS ndom1
DEF ndom4 = ndom2 MINUS ndom1
DEF ndom5 = ndom2 MULT ndom1
DEF ndom6 = ndom2 DIV ndom1
DEF ndom7 = ndom2 REM ndom1
DEF ndom8 = NEG ndom2
DEF tdom1 = ndom1 EQ ndom2
DEF tdom2 = ndom1 LT ndom2
DEF tdom3 = ndom1 LE ndom2
DEF tdom4 = ndom1 GT ndom2
DEF tdom5 = ndom1 GE ndom2
DEF tdom6 = ndom1 NE ndom2

END TesteScript

--------------------------------------

module  TesteScript  where

import  Prel_Tav

ndom1 = 15
```

```
ndom2 = 20
ndom3 = ndom2 + ndom1
ndom4 = ndom2 - ndom1
ndom5 = ndom2 * ndom1
ndom6 = ndom2 `div` ndom1
ndom7 = ndom2 `mod` ndom1
ndom8 = negate ndom2
tdom1 = ndom1 ==  ndom2
tdom2 = ndom1 < ndom2
tdom3 = ndom1 <= ndom2
tdom4 = ndom1 > ndom2
tdom5 = ndom1 >= ndom2
tdom6 = ndom1 /= ndom2

---------------------------------------
```

2. Exemplo de programa com os operadores binários e unários que operam sobre o domínio de valores lógicos ($\mathcal{N}$)

```
---------------------------------------

MODULE TesteScript

DOMAINS    Tdom = T

DEFINITIONS

DEF tdom1 = "TT"
DEF tdom2 = "FF"
DEF tdom3 = tdom1 AND tdom2
DEF tdom4 = tdom1 OR tdom2
DEF tdom5 = NOT tdom1

END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

tdom1 = True
tdom2 = False
tdom3 = tdom1   &&   tdom2
tdom4 = tdom1    ||  tdom2
tdom5 = not  tdom1

---------------------------------------
```

3. Exemplo de programa com construção condicional

```
--------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T

DEFINITIONS

DEF ndom1 =  11
DEF t1 =  "FF"
DEF f1 (x: Ndom) : Ndom = x EQ 0 -> 1, 2

END TesteScript

--------------------------------------
```

```
module  TesteScript  where

import  Prel_Tav

ndom1 = 11
t1 =  False
f1  x  = if  x  ==   0  then  1  else  2

--------------------------------------
```

4. Exemplo de programa com construção de criação e atualização de listas

```
--------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T ;
           LdomT = T* ;
           LdomN = N*

DEFINITIONS

DEF ndom1 = 2
DEF ndom2 = 100
DEF t1 = "FF"
DEF ldomT1 = <"FF", "TT", "FF">
DEF ldomT2 = ldomT1 {ndom1 = t1}
DEF ldomN1 = <ndom1 .. ndom2>
DEF ldomN2 = <15 .. 45>

END TesteScript

--------------------------------------
```

```
module  TesteScript  where

import  Prel_Tav

ndom1 =  2
ndom2 =  100
t1 =  False
ldomT1 =  [ False ,  True ,  False ]
ldomT2 =  atualiza_lista  ldomT1  [ ( ndom1 , t1 ) ] 1
ldomN1 =  [ ndom1 .. ndom2 ]
ldomN2 =  [ 15 .. 45 ]

---------------------------------------
```

5. Exemplo de programa com construção de criação de listas usando compreensão de listas

```
---------------------------------------
MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T ;
           LdomN = N*

DEFINITIONS

DEF ndom1 =  6
DEF ndom2 =  10
DEF t1 =  "TT"
DEF ldomN' =  <1 .. 5>
DEF ldomN'' =  <ndom1 .. ndom2>
DEF ldomN1 =  < n' PLUS n'' | n' <- ldomN' :: t1 | n'' <- ldomN'' >

END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

ndom1 =  6
ndom2 =  10
t1 =  True
ldomN' =  [ 1 .. 5 ]
ldomN'' = [ ndom1 .. ndom2 ]
ldomN1 = [  n'   +  n''  |  n'  <-  ldomN' ,  t1  ,  n''  <-  ldomN'' ]
```

6. Exemplo de programa com operadores que operam sobre listas

```
---------------------------------------
```

```
MODULE TesteScript

DOMAINS    Ndom = N ;
           LdomN = N*

DEFINITIONS

DEF ndom1 = 4
DEF ndom2 = 9
DEF ndom3 = 3
DEF ldomN1 = <ndom1 .. ndom2>
DEF ldomN2 = ndom3 PRE ldomN1
DEF ldomN3 = <1,2,3>
DEF ldomN4 = <10>
DEF ldomN5 = ldomN3 CAT ldomN1 CAT ldomN4
DEF ndom5 = SIZE ldomN5
DEF ndom6 = ldomN1 EL 3
DEF q1 = "ban"
DEF q2 = "ana"
DEF q3 = q1 CAT q2

END TesteScript
```

```
module  TesteScript  where

import  Prel_Tav

ndom1 =   4
ndom2 =   9
ndom3 =   3
ldomN1 =  [ ndom1 .. ndom2 ]
ldomN2 =  ndom3    :  ldomN1
ldomN3 =  [ 1 ,  2 ,  3 ]
ldomN4 =  [ 10 ]
ldomN5 =  ldomN3  ++  ldomN1  ++  ldomN4
ndom5 =   length  ldomN5
ndom6 =  ldomN1 !!  3
q1 =  "ban"
q2 =  "ana"
q3 =  q1  ++  q2
```

---

7. Exemplo de programa com construção IS

```
--------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T

DEFINITIONS

DEF ndom1 =  1
DEF ndom2 =  2
DEF tdom1 = ndom1 IS 3

END TesteScript

--------------------------------------

module  TesteScript  where

import  Prel_Tav

ndom1 =   1
ndom2 =   2
tdom1 =   case  ndom1  of
              3  -> True
              otherwise -> False

--------------------------------------
```

8. Exemplo de programa com construção CASE

```
--------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T ;
           Tupladom = N

DEFINITIONS

DEF ndom1 =  1
DEF ndom2 =  2
DEF ndom3 = CASE ndom1 PLUS ndom2
                 /3 -> ndom1
                 /5 -> ndom2
                END
```

```
END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

ndom1 =   1
ndom2 =   2
ndom3 =   case  ndom1   +  ndom2  of
              3  ->  ndom1
              5  ->  ndom2

---------------------------------------
```

9. Exemplo de programa com construção LET/IN

```
---------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T;
           Ldom = Ndom*;
           a : Ndom;
           b : Ldom

DEFINITIONS

DEF n1 = 11
DEF t1 = "TT"
DEF n2 =
    LET ndom1 =  11
    LET ndom2 =  ndom1
    LET ldom1 =  < ndom1, ndom2 >
    LET ldom3  = ldom1
    LET n11 = 11
    LET t11 = "TT"
    LET b = <1,2,3>
    LET ldom2 =  b
    IN  n1 PLUS n11
DEF t2 = "FF"

END TesteScript

---------------------------------------
```

```
module  TesteScript  where

import  Prel_Tav

n1 =   11
t1 =   True
n2 =    let ndom1 =   11     in
   let  ndom2 =   ndom1     in
   let  ldom1 =   [ ndom1 ,  ndom2 ]     in
   let  ldom3 =   ldom1     in
   let  n11 =   11     in
   let  t11 =   True     in
   let  b =   [ 1 ,  2 ,  3 ]     in
   let  ldom2 =   b     in
  n1   +  n11
t2 =   False

---------------------------------------
```

10. Exemplo de programa com definição e aplicação de funções

```
---------------------------------------

MODULE TesteScript

DOMAINS    Ndom = N ;
           Tdom = T

DEFINITIONS

DEF ndom1 = 11
DEF q1 = "teste"
DEF f1 (x1: N) : Q = q1
DEF q2 = f1 11

END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

ndom1 =   11
q1 =   "teste"
f1  x1  =   q1
q2 =  f1 11

---------------------------------------
```

11. Exemplo de programa com atribuições simples, construção condicional e tuplas

```
---------------------------------------

MODULE TesteScript

DOMAINS    Adom = (N, tdom1, y:N);
           Ndom = N ;
           Tdom = T

DEFINITIONS

DEF adom1 =  (11, t1, ndom1)
DEF ndom1 =  11
DEF q1 =  "teste"
DEF t1 =  "FF"
DEF f2 (x: Ndom) : Ndom = x EQ 0 -> 0 PLUS 2, 1
DEF f1 (x: Ndom) : Ndom = x EQ 1 -> 1, 0

END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

adom1 =   (11, t1, ndom1)
ndom1 =   11
q1 =   "teste"
t1 =   False
f2  x  =  if  x    ==   0   then   0   +  2   else   1
f1  x  =  if  x    ==   1   then   1   else   0

---------------------------------------
```

12. Exemplo de programa com tuplas e funções associadas a domínios

```
---------------------------------------

MODULE TesteScript

DOMAINS Adom = (m:N, T, y:N, a:T) ;
        Bdom = (m:N, T, y:N);
        Ndom = N ;
        Tdom = T

DEFINITIONS

DEF adom1 = (11, t1, ndom1, t1)
DEF bdom1 = (11, t1, ndom1)
DEF ndom1 = 11
DEF ndom2 = 4
```

```
DEF ndom3 = ndom1 PLUS ndom2
DEF t1 = "FF"
DEF Adom.f1 (x: Ndom) : Ndom = THIS.y PLUS x
DEF g1 (b: Bdom) (a: Bdom) (x: Ndom) : Ndom = b.m PLUS b.y
DEF ndom4 = g1 adom1 bdom1 ndom1 REM g1 adom1 bdom1 ndom1

END TesteScript

---------------------------------------

module  TesteScript  where

import  Prel_Tav

adom1 =   (11, t1, ndom1, t1)
bdom1 =   (11, t1, ndom1)
ndom1 =   11
ndom2 =   4
ndom3 =   ndom1   +  ndom2
t1 =   False
f1__Adom this__1  x  =  (let ( a__1,  a__2,  a__3,  a__4) =  this__1  in  a__3 )    +  x
g1  b   a   x  =  (let ( a__1,  a__2,  a__3) =  b  in  a__1 )    +
                  (let ( a__1,  a__2,  a__3) =  b  in  a__3 )
ndom4 =  g1 (let (  param__1 ,  param__2 ,  param__3 ,  param__4 ) = adom1 in
         (  param__1 ,  param__2 ,  param__3 ) )  bdom1   ndom1 `mod`
         g1 (let (  param__1 ,  param__2 ,  param__3 ,  param__4 ) =
         adom1 in (  param__1 ,  param__2 ,  param__3 ) )  bdom1   ndom1

---------------------------------------
```

13. Exemplo de programa com tuplas e funções associadas a domínios

```
---------------------------------------

MODULE TesteScript

DOMAINS Adom = (m:N, T, y:N, n:T) ;
        Ndom = N

DEFINITIONS

DEF ndom1 = 10
DEF adom1 = (11, "TT", 11, "TT")
DEF Adom.f (x:Ndom) : Ndom = x
DEF n1 = adom1.f (10) PLUS adom1.f (10)

END TesteScript

---------------------------------------

module  TesteScript  where
```

```
import  Prel_Tav

ndom1 =   10
adom1 =   (11, True, 11, True)
f__Adom this__1  x  =   x
n1 =   f__Adom adom1   10  +  f__Adom adom1   10

---------------------------------------
```

Execução no interpretador $\mathcal{HASKELL}$:

```
Hugs session for:
/pkg/cursos/haskell/hugs/hugs/lib/Prelude.hs
Prel_Tav.hs
ex13.hs
Type :? for help
TesteScript> ndom1
10
TesteScript> adom1
(11, True, 11, True)
TesteScript> f__Adom adom1 ndom1
10
TesteScript> n1
20
TesteScript> :q
[Leaving Hugs]

---------------------------------------
```

14. Exemplo de programa com atribuição de tuplas de tamanhos diferentes

```
---------------------------------------

MODULE TesteScript

DOMAINS Adom = (m:N, T, y:N) ;
        Bdom = Adom EXT (a:N) ;
        Ndom = N

DEFINITIONS

DEF bdom1 = (11, "TT", 11, 11)
DEF adom1 = bdom1

END TesteScript

---------------------------------------
```

```
module  TesteScript  where

import  Prel_Tav

bdom1 =   (11, True, 11, 11)
adom1 =  let (  a__1 ,  a__2 ,  a__3 ,  a__4 ) = bdom1 in
         ( a__1 ,  a__2 ,  a__3 )
```

Execução no interpretador $\mathcal{HASKELL}$:

```
Hugs session for:
/pkg/cursos/haskell/hugs/hugs/lib/Prelude.hs
Prel_Tav.hs
ex14.hs
Type :? for help
TesteScript> bdom1
(11, True, 11, 11)
TesteScript> adom1
(11, True, 11)
TesteScript> :q
[Leaving Hugs]
```

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