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

Implementação de um Gerador de Interpretadores de uso Geral

Fernando Magno Quintão Pereira

Orientador: Roberto da Silva Bigonha

Co-Orientadora: Mariza A. S. Bigonha

Relatório Técnico do
Laboratório de Linguagens de Programação

LLP004/2001

Av. Antônio Carlos, 6627

31270-010 - Belo Horizonte - MG

26 de Agosto de 2003

Conteúdo

1 Introdução **2**

2 Semântica Formal **3**
2.1 Semântica Denotacional 4
2.2 Exemplo de Descrições Semânticas 5

3 A Linguagem *SCRIPT* **7**
3.1 Estrutura de Módulos em *SCRIPT* 9
3.1.1 Módulo PROJECT 10
3.1.2 Módulo SYNTAX 10
3.1.3 Módulo MODULE 11

4 O Papel da Linguagem *Haskell* no Projeto **11**

5 Diferenças Conceituais entre *Haskell* e *Script* **12**
5.1 Tuplas e Extensões de Tuplas 12
5.2 Funções Associadas a Domínios 14

6 Soluções Propostas para Problemas de Tradução **15**
6.1 O Tipo Universal 15
6.2 Estratégias para Manipulação de Tuplas 18
6.3 Estratégias para Tradução de Funções 21
6.4 Análise das Soluções Propostas 23

7 Geração de Interpretadores e Interpretação de Programas **24**
7.1 Primeira Etapa – Geração de Interpretadores 26
7.2 Segunda Parte – Geração de Tradutores 28
7.3 Terceira Parte – Geração de Árvore de Sintaxe 29
7.4 Quarta Parte – Interpretação de Programas 32

8 Princípios de Funcionamento dos Interpretadores Gerados **32**

9 Descrição Formal da Linguagem *LiLoCa* **34**
9.1 Definição das Unidades Léxicas da Linguagem 35
9.2 A Gramática da Linguagem *LiLoCa* 35
9.3 Descrição Semântica da Linguagem *LiLoCa* 37
9.4 Exemplo de programa escrito em *LiLoCa* 44

10 Conclusão **45**

Lista de Figuras

1 Programa *SCRIPT* para somas de números inteiros. 8
2 Gramática da linguagem *Arit* 9
3 Descrição Semântica da Linguagem *Arit* 10
4 Programa *SCRIPT* incorreto 16
5 Programa *Haskell* incorreto 17
6 Mínima Definição do Tipo Universal 17
7 Atualização de tuplas 21
8 Inclusão de funções ao Tipo Universal 22
9 Declaração de funções pertencentes ao Tipo Universal 22
10 Geração automática de interpretadores e interpretação 26
11 A geração de interpretadores 27
12 Arquivo `Main.hs` 27
13 Arquivo `Universal.hs` 28
14 Tabela de símbolos gerada para *Arit* 28
15 Analisador léxico gerado para a linguagem *Arit* 29
16 Analisador léxico gerado para a linguagem *Arit* 30
17 Geração de tradutores 30
18 Árvore sintática que representa o programa *1 + 2 + 3* 31
19 módulo `Program` que contém uma representação da árvore de sintaxe para o programa *1 + 2 + 3* 31
20 Tradução de programas para formato de árvore sintática 32
21 Relação entre os arquivos *Haskell* gerados 33
22 Sequência de interpretação para *1 + 2 + 3* 34
23 Programa *SCRIPT* que descreve o alfabeto da linguagem *LiLoCa*. 35
24 Definição semântica de fatores de uma expressão 39
25 Definição semântica de termos de expressões 40
26 Definição semântica de expressões aritméticas 41
27 Definição semântica de expressões booleanas 41
28 Definição semântica de expressões 42
29 Definição semântica de comandos 43
30 Definição semântica de declarações 43
31 Definição semântica do corpo de funções 43
32 Definição semântica de um programa escrito em *LiLoCa* 44
33 Definição semântica de seqüências de comandos e declarações 44
34 Exemplo de programa escrito em *LiLoCa* 45

Resumo

Este trabalho contém a descrição de uma ferramenta capaz de gerar interpretadores para linguagens de programação a partir da descrição dessas linguagens via programas $\mathcal{SCRIPT}$. $\mathcal{SCRIPT}$ é uma linguagem funcional cujo propósito principal é prover uma notação adequada para a descrição dos aspectos léxicos, sintáticos e semânticos de linguagens de programação. Os interpretadores automaticamente produzidos por esta ferramenta são programas codificados em *Haskell*, uma outra linguagem pertencente ao paradigma funcional.

Palavras Chaves: Interpretador, Script, Haskell, Linguagem Funcional, Tradutor, Geração Automática de Código, Semântica Denotacional

1 Introdução

O trabalho apresentado neste relatório faz parte de um projeto maior, que envolve professores e estudantes pesquisadores do Laboratório de Linguagens de Programação da UFMG e que se desenvolve em torno da linguagem $\mathcal{SCRIPT}$ [2].

O objetivo deste projeto é desenvolver um ambiente de programação que permita a geração de interpretadores com base na descrição semântica de linguagens de programação e que permita a interpretação de programas escritos nestas linguagens. Tais interpretadores serão gerados a partir da análise de um programa escrito na linguagem $\mathcal{SCRIPT}$. Tal programa contém a descrição dos aspectos léxico, sintático e semântico da linguagem alvo.

A linguagem $\mathcal{SCRIPT}$ está presente em outros trabalhos realizados no Laboratório de Linguagens de Programação, dentre os quais pode-se citar:

- O projeto e implementação de um compilador de $\mathcal{SCRIPT}$ para *Lamb*, uma versão estendida do cálculo λ, a linguagem utilizada como código intermediário na compilação de linguagens funcionais [3].
- Um tradutor de definições de funções escritas em $\mathcal{SCRIPT}$ para *Haskell* [4].

O restante deste documento contém uma breve explicação sobre as atividades relacionadas ao projeto do Gerador de Interpretadores desenvolvidas no decorrer do segundo semestre de 2001. Algumas seções descrevem o produto implementado e o seu funcionamento, enquanto outras cuidam do arcabouço teórico que fundamentou esta implementação. Estas seções estão organizadas da seguinte forma:

- Na Seção 2 é apresentada uma breve visão sobre os conceitos relacionados à semântica formal de linguagens de programação, a principal ferramenta teórica de suporte ao sistema desenvolvido neste trabalho.
- A Seção 3 descreve a linguagem $\mathcal{SCRIPT}$ e define a linguagem *Arit*, que será utilizada como exemplo em todo este relatório.
- A Seção 4 contém uma breve descrição sobre a linguagem *Haskell* e seu papel no sistema de programação desenvolvido.
- A Seção 5 ressalva as principais diferença entre as linguagens *Haskell* e $\mathcal{SCRIPT}$. Como a geração automática de interpretadores envolve a tradução de programas escritos em $\mathcal{SCRIPT}$ para *Haskell*, é interessante que tais diferenças sejam bem compreendidas, a fim de que fiquem claras as razões que motivaram as soluções adotadas no processo de tradução.

- Na Seção 6 são mostradas algumas das estratégias de tradução adotadas para contornar as diferenças entre as linguagens $\mathcal{SCRIPT}$ e *Haskell* relacionadas na Seção 5.

- A Seção 7 contém uma breve descrição do Ambiente de Programação desenvolvido neste trabalho. Tais descrições abrangem o processo de geração de interpretadores e interpretação e também as decisões de implementações adotadas.

- A Seção 8 descreve os princípios de funcionamento dos interpretadores gerados automaticamente.

- A Seção 9 contém um tutorial. Neste tutorial é mostrado um programa $\mathcal{SCRIPT}$ que descreve uma linguagem simples, porém rica de recursos e funcionalidades.

- finalmente, a Seção 10 apresenta a conclusão sobre os trabalhos desenvolvidos, e sugere possíveis linhas de pesquisa que podem ser desenvolvidas no futuro.

2 Semântica Formal

A Semântica Formal é um conjunto de técnicas, formalismos e notações utilizadas para descrever de modo preciso o significado das construções sintáticas de uma linguagem de programação. Por construções sintáticas, entende-se, por exemplo, cada um dos comandos, expressões e declarações que são utilizados para escrever programas em uma linguagem de programação específica. Assim, enquanto a gramática especifica a estrutura física que terão os programas escritos em uma certa linguagem, as descrições semânticas determinam como se comportarão aqueles programas, ou seja, que resultados produzirão para cada possível conjunto de dados de entrada.

Embora possua diversos propósitos, o estudo de Semântica Formal tem recebido ao longo dos anos uma atenção muito menor que outros aspectos da teoria das linguagens de programação, como por exemplo a sintaxe. Ainda assim, muitas das idéias que sustentam esta teoria existem já desde longa década. Foi, contudo, a partir dos trabalhos de Dana Scott [9], no início da década de 70, que se solidificaram a maior parte dos conceitos atualmente utilizados na área de semântica formal de linguagens de programação.

A despeito de ser uma área teórica relativamente pouco estudada, em comparação com outros aspectos da teoria das linguagens de programação, a Semântica Formal se presta a diversos propósitos:

1. Prover conceitos precisos e independentes de implementações específicas.
2. Prover técnicas de especificação não ambíguas.
3. Prover uma base teórica para suportar provas a respeito do comportamento de programas.

Um dos primeiros papéis a que se prestam especificações semânticas é prover descrições precisas sobre uma linguagem e que sejam independentes do dispositivo computacional. O significado essencial de qualquer construção de uma linguagem de programação claramente não pode depender de características de uma máquina particular, dada a grande variedade de diferentes tecnologias disponíveis, e dado o caráter mutável das mesmas. Assim, o mais coerente, portanto, seria que a partir de um mesmo programa fonte fossem gerados códigos executáveis que, mesmo em diferentes computadores, produziriam os mesmos resultados, quando submetidos aos mesmos dados de entrada. Isto contudo, nem sempre se verifica.

Além de serem independentes dos dispositivos computacionais, as descrições semânticas precisam ser não ambíguas, de modo que, a partir de um código fonte, apenas um significado possa ser inferido. Seja, por exemplo, a descrição informal de um comando de atribuição, tal qual pode ser encontrado no relatório oficial a respeito da linguagem Pascal [10]:

Ide := Exp *O comando de atribuição substitui o valor corrente de uma variável por um novo valor, especificado via uma expressão.*

Dada a informalidade da descrição mostrada, esta apresenta alguns problemas quanto à clareza. Por exemplo, para alguém que pouco conhece sobre Ciência da Computação, poderia não ser claro o conceito de "variável" e de "valor corrente" de uma variável. Até mesmo para um cientista da computação não fica claro, naquela descrição informal, como um valor é especificado por uma expressão. A semântica formal visa, justamente prover descrições precisas sobre o significado de uma construção como o comando de atribuição, anteriormente visto. Funções matemáticas são a principal ferramenta utilizada para que tal objetivo seja alcançado.

É importante também dizer que as técnicas de semântica formal permitem que sejam enunciados e provados com rigor matemático muitas propriedades importantes das linguagens de programação. Dessa forma, a fim de provar que um determinado programa funciona satisfatoriamente é preciso mostrar que seu significado real coincide com o seu significado pretendido, e isto exige algum formalismo.

2.1 Semântica Denotacional

A Semântica Denotacional é o principal ferramental teórico utilizado no contexto deste trabalho. Denotações são entidades matemáticas abstratas que modelam o

significado de construções das linguagens de programação. Por exemplo, o número denotado pelo algarismo 6 pode ser também denotado por diferentes expressões: 4 + 2, 12 - 6, 2 × 2, etc.

A Semântica Denotacional é apenas um ramo da Semântica Formal. Existem outras subdivisões desta área, que se prestam a outros tipos de especificações. Dentre os exemplos disponíveis, pode-se citar: *Semântica Operacional*, *Semântica Axiomática* e a *Semântica algébrica*. Estas subdivisões não constituem maneiras alternativas de fazer o que a Semântica Denotacional já faz, pois cada uma delas tem seus próprios objetivos e lança mão de ferramentas particulares.

2.2 Exemplo de Descrições Semânticas

A Semântica Denotacional emprega funções matemáticas e um tipo especial de conjunto, denominado domínio, para prover descrições não ambíguas de construções de linguagens de programação. Via estes dois conceitos, de domínios e de funções, é possível modelar o estado da máquina, que é definido pela sua memória, os dados na entrada e os resultados gerados na saída. Assim, uma soma de duas expressões, sintaticamente representada como em 1, pode, informalmente ser descrita como: *o valor de* $E_1 + E_2$ *é a soma numérica dos valores de* E_1 *e* E_2 *se estas duas subexpressões denotam números. Se qualquer uma delas representa um valor que não seja um número, então sua soma representa um erro.* A expressão 1 pode também ser descrita de maneira mais formal, como na Equação 2.

$$\mathbf{E_1 + E_2} \tag{1}$$

$$\begin{aligned}\mathbf{E[E_1 + E_2]s} \;=\; & (\mathbf{E[E_1]s = (v_1,\, s_1)}) \rightarrow \\ & ((\mathbf{E[E_2]s = (v_2,\, s_2)}) \rightarrow \\ & (\mathbf{isNum\, v_1 \;\; and \;\; isNum\, v_2} \rightarrow \\ & (\mathbf{v_1 + v_2,\, s_2}),\, \mathbf{error}),\, \mathbf{error}),\, \mathbf{error}\end{aligned} \tag{2}$$

Para que a Equação 2 possa ser completamente entendida, é preciso que alguns conceitos, até agora apenas referenciados, sejam melhor explicados. Em primeiro lugar, *domínios* são um tipo especial de conjuntos. Todo domínio contém um tipo particular de elemento usado para representar indefinições. Por *indefinições* pretende-se nomear qualquer entidade que não possua uma representação específica em um domínio. Por Exemplo, se considerarmos que o nome **Num** denota o domínio dos números inteiros, então teremos que $\mathbf{\{1,\, 2,\, 3,\, \ldots\}}$ são todos elementos de **Num**, como também é um elemento de **Num** o elemento indefinido (**unbound**). Embora fuja ao escopo deste relatório explicar porque domínios são definidos desta

forma, pode-se dizer que esta estrutura permite que domínios sejam capazes de conter:

- Definições recursivas de funções.
- Definições recursivas de conjuntos.

O estado de um dispositivo computacional, por exemplo, pode ser representado como um domínio de tuplas de três termos, como pode ser visto nas Equações 3, 4, 5, 6 e 7. Assim, a Equação 3 significa que o estado de uma máquina é definido pelo conteúdo armazenado na memória da mesma, pelos valores enfileirados na sua entrada de dados e pelos valores gerados na saída, como resultado de alguma computação. **Memory**, por sua vez, é o nome dado ao domínio que representa a memória do computador. Este é o domínio de funções que, dado um identificador de área de memória (**Ide**), retornam o valor armazenado naquela posição. Tanto a entrada (**Input**) quanto a saída (**Output**) de dados, por outro lado, são modeladas como domínios que denotam filas de valores. Valores (**Value**), finalmente, no contexto de uma máquina muito simples, podem ser, ou números inteiros (**Num**), ou valores booleanos (**Bool**).

$$\begin{aligned}
\mathbf{State} &= \mathbf{Memory} \times \mathbf{Input} \times \mathbf{Output} && (3)\\
\mathbf{Memory} &= \mathbf{Ide} \rightarrow [\mathbf{Value} + \mathbf{unbound}] && (4)\\
\mathbf{Input} &= \mathbf{Value}* && (5)\\
\mathbf{Output} &= \mathbf{Value}* && (6)\\
\mathbf{Value} &= \mathbf{Num} + \mathbf{Bool} && (7)\\
& && (8)
\end{aligned}$$

Funções Semânticas Tais funções são, junto com os domínios, o principal ferramental utilizado para descrever o funcionamento de construções de linguagens de programação. A fim de melhor elucidar este conceito, será definida uma função para denotar o significado de expressões simples. *Expressões* podem ser entendidas de modo diverso, dependendo-se do paradigma de programação considerado. Assim, em linguagens do paradigma *funcional*, expressões denotam valores. No paradigma *imperativo*, por outro lado, expressões também denotam valores, mas podem, excepcionalmente, significar uma mudança no estado da máquina. A Equação 9 contém a definição de uma função (**E**) utilizada para descrever o significado de expressões em uma linguagem do paradigma imperativo.

$$\mathbf{E} : \mathbf{Exp} \rightarrow [\mathbf{State} \rightarrow [\mathbf{Value} \times \mathbf{State}] + \mathbf{error}] \qquad (9)$$

A Equação 9, dada a forma como está declarada, pode denotar três fatos importantes:

- A possibilidade de expressões causarem erros, por exemplo: $\mathbf{1} + \mathbf{true}$.
- O papel do estado da máquina na determinação do valor de algumas expressões. Assim, o valor da expressão $\mathbf{x} + \mathbf{1}$ depende de qual valor está associado à variável **x** na memória do computador. Da mesma forma, uma expressão como $\mathbf{1} + \mathbf{read}()$ depende do conteúdo da entrada de dados.
- A possibilidade de que a avaliação de uma certa expressão possa mudar o estado da máquina. Por exemplo, a expressão **read()** retorna o primeiro valor na entrada padrão de dados de um computador, e remove aquele valor de lá.

Deve ser definida uma função **E** para cada possível expressão de uma linguagem de programação. Tal função **E** deve ser capaz de especificar o significado de *constantes*, *variáveis*, *operações unárias* e *operações binárias*, conforme estas expressões possam ou não ocorrer na linguagem de programação em questão.

Estando já definidos os conceitos de domínios e expressões, pode-se tornar a analisar a Equação 2 de modo mais detalhado. Neste caso, a construção $\mathbf{b} \rightarrow \mathbf{v_1}, \mathbf{v_2}$ significa: *Se o valor denotado por* **b** *for verdadeiro, então* $\mathbf{v_1}$ *senão* $\mathbf{v_2}$. Assim, uma soma de duas subexpressões apenas estará correta se cada uma destas subexpressões denotarem valores inteiros, isto é, pertencentes ao domínio dos números inteiros (**Num**). Neste caso, o valor denotado por uma soma de duas subexpressões é a soma aritmética de cada uma destas subpartes.

Descrições semânticas, como a apresentada na Equação 2 são suficientemente precisas para poderem ser fornecidas como instruções para programas de computador. Assim, a partir da descrição semântica de uma linguagem de programação qualquer é possível gerar um interpretador para aquela linguagem. Este é o objetivo básico deste trabalho: gerar interpretadores para linguagens de programação a partir da descrição semântica das mesmas.

3 A Linguagem $\mathcal{SCRIPT}$

$\mathcal{SCRIPT}$ é uma linguagem funcional, com algumas características do paradigma de Orientação a Objetos. A linguagem $\mathcal{SCRIPT}$ foi idealizada pelo professor Roberto Bigonha [2] e seu propósito principal é descrever outras linguagens de programação. A partir da descrição $\mathcal{SCRIPT}$ de uma linguagem $\mathcal{L}$ é possível gerar interpretadores e analisadores léxicos e sintáticos para $\mathcal{L}$, sendo este o objetivo principal do projeto.

```
GRAMMAR Arit
SYNTAX
  exp   ::=     exp "+" exp
        |       num
LEXIS
  UNIT ::= num
  num  ::= digit + : NUMBER;
  digit === "0" .. "9"
END Arit
```

Figura 1: Programa $\mathcal{SCRIPT}$ para somas de números inteiros.

Algumas características importantes da linguagem $\mathcal{SCRIPT}$ estão enumeradas logo abaixo:

- Linguagem do paradigma funcional.
- Equivalência estrutural de tipos.
- Linguagem altamente modular.
- Dotada de mecanismos de controle de visibilidade e encapsulação.
- Dotada de características de linguagens orientadas a objetos:
  - mecanismo de herança;
  - funções sobrecarregadas;
  - associação (binding) dinâmica.

Uma descrição $\mathcal{SCRIPT}$ completa engloba os três aspectos principais de uma linguagem: léxico, sintático e semântico. Dada a grande modularidade de $\mathcal{SCRIPT}$, cada um destes aspectos pode ser especificado em um arquivo diferente, ou no mesmo arquivo, ou ainda em vários arquivos separados.

A título de ilustração, a Figura 1 contém a descrição dos aspectos léxicos e sintáticos de uma linguagem muito simples, denominada *Arit*, capaz de reconhecer somas de números inteiros. A descrição dos símbolos reconhecidos por esta linguagem encontra-se no módulo denominado `LEXIS` ao passo que a gramática que a define está especificada no módulo identificado pelo rótulo `SYNTAX`. O conjunto de símbolos reconhecidos por essa linguagem é formado somente por seqüências de números inteiros e pelo sinal de adição (+).

No módulo `LEXIS` está definido o conjunto de símbolos reconhecido pela linguagem, ou seja, o seu alfabeto. Caso estes símbolos sejam agrupados em uma

exp	→	exp + exp
	|	`num`

Figura 2: Gramática da linguagem *Arit*

certa seqüência, podem vir a formar um programa correto. Os símbolos de uma linguagem formam um programa correto quando estão agrupados em uma ordem que concorda com a sua gramática. Toda linguagem de programação possui uma gramática bem definida que define quais seqüências de símbolos formam programas válidos. A gramática que define a estrutura sintática da linguagem *Arit* pode ser vista na Figura 2. Observe que todas as regras de derivação da gramática aparecem no módulo `SYNTAX` do programa da Figura 1. Neste exemplo, a gramática da linguagem *Arit* possui somente duas regras de derivação.

Além da descrição dos aspectos léxico e sintático de uma linguagem, para que sua especificação esteja completa, é preciso que seja descrito também o significado de cada um de seus possíveis comandos e expressões. Este é o aspecto semântico de uma linguagem de programação. Tais descrições, em $\mathcal{SCRIPT}$ são feitas por meio de declarações de tipos e funções que descrevem as ações que serão realizadas por cada construção correta da linguagem. O programa da Figura 3 contém as funções semânticas que descrevem a linguagem *Arit*. Este programa formaliza o significado de cada construção da linguagem *Arit*. Nesta linguagem, uma expressão é um número inteiro ou uma soma de duas expressões e, assim, um programa escrito em *Arit* consiste em uma seqüência de somas de expressões númericas. A partir dos dois programas $\mathcal{SCRIPT}$ que foram apresentados é possível ter uma idéia perfeita acerca do funcionamento e da organização da linguagem *Arit*.

3.1 Estrutura de Módulos em $\mathcal{SCRIPT}$

A estrutura completa de uma definição $\mathcal{SCRIPT}$ é formada por um módulo principal e um 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 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 $\mathcal{SCRIPT}$: PROJECT, SYNTAX e MODULE.

```
MODULE Arit

DOMAINS
    Exp = [Exp "+" Exp] | [N]

DEFINITIONS

DEF elab-exp (exp) : N =
        CASE exp
            /[exp1 "+" exp2] ->
                LET exp1' = elab_exp (exp1)
                LET exp2' = elab_exp (exp2)
                IN exp1' PLUS exp2'
            /[n] -> n
        END

END Arit
```

Figura 3: Descrição Semântica da Linguagem *Arit*

3.1.1 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, a qual 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.

3.1.2 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.

3.1.3 Módulo MODULE

Os módulos MODULE permitem encapsular definições de domínios e funções podendo também ser usados para estabelecer a interface de comunicação entre os módulos. Tais módulos são compostos 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.

4 O Papel da Linguagem *Haskell* no Projeto

O interpretador que pode ser obtido via uma descrição $\mathcal{SCRIPT}$ é gerado a partir das funções semânticas que aparecem no módulo denominado `MODULE`, presente naquela descrição. Infeliz- mente, porém, um programa $\mathcal{SCRIPT}$ não pode ser diretamente compilado para linguagem de máquina e executado, pois ainda não existe um compilador deste tipo. Desta forma, a fim de serem executados, os programas escritos em $\mathcal{SCRIPT}$ precisam ser traduzidos para alguma outra linguagem que possua um compilador ou um interpretador operacional, como, por exemplo, as linguagens *C* [13] ou *Java* [11].

A linguagem de programação escolhida como alvo neste processo de tradução chama-se *Haskell* [7]. A principal razão que levou à escolha de *Haskell* foi o fato desta ser uma linguagem pertencente ao paradigma funcional, assim como $\mathcal{SCRIPT}$. Caso fosse definida alguma linguagem imperativa, por exemplo: *Pascal* [10] ou *C* [13], como alvo no processo de tradução, haveria grande dificuldade, pois estas linguagens são muito diferentes daquelas chamadas funcionais.

Além de muito semelhante a $\mathcal{SCRIPT}$ a linguagem *Haskell* possui algumas vantagens que são típicas de linguagens funcionais, por exemplo:

- Avaliação *Lazy*, isto é, os parâmetros de uma função são avaliados apenas quando necessários no corpo daquela função.

- Polimorfismo universal paramétrico, ou seja, a capacidade de definir funções que se comportam da mesma maneira para parâmetros de tipos diferentes.

- Funções de ordem superior, isto é, funções que podem ser passadas como parâmetros para outras funções, ou retornadas como o resultado de alguma função.

- Funções parciais, ou seja, uma função que, quando aplicada a apenas alguns de seus argumentos, retorna uma outra função, mais específica, que pode receber os argumentos restantes da função original.

- Estruturas de dados de tamanho infinito, como por exemplo listas. Esta é uma decorrência direta da avaliação *Lazy*, pois embora tais estruturas não sejam "infinitas" de fato, esta forma de avaliação de parâmetros permite ao programador enxergá-las desta maneira.

- Permite a prova da corretude de programas de forma mais fácil, em grande parte em decorrência da ausência de "efeitos colaterais", isto é, mudanças de estado durante a avaliação de expressões.

- Ausência de variáveis globais e desvios incondicionais, que, quando usados sem disciplina, contribuem para a produção de códigos ilegíveis.

Além dessas características, *Haskell* possui outras que a tornam uma linguagem de programação "limpa" e bem estruturada, adequada às provas de corretude de programas e à geração de código reutilizável. A linguagem ainda dispõe de classes de tipos e funções sobrecarregadas, estas últimas, duas características típicas de linguagens orientadas a objetos.

5 Diferenças Conceituais entre *Haskell* e *Script*

Embora *Haskell* e $\mathcal{SCRIPT}$ sejam duas linguagens pertencentes ao paradigma funcional, elas possuem inúmeras diferenças. As diferenças sintáticas entre as duas linguagens não representam grande problema para o processo de tradução. Existem, entretanto, algumas diferenças conceituais que requerem uma análise mais elaborada.

5.1 Tuplas e Extensões de Tuplas

Em primeiro lugar, $\mathcal{SCRIPT}$ possui os conceitos de domínios de tipos e de extensão de domínios que estão ausentes em *Haskell*. A Equação 10, por exemplo, define A como o domínio de todas as tuplas que possuem no mínimo dois campos inteiros. Dessa forma, qualquer uma das Equações 11, 12 ou 13 é verdadeira na linguagem $\mathcal{SCRIPT}$.

$$Def\ A = (N, N) \tag{10}$$

$$(1, 2) \in A \tag{11}$$

$$(1, 2, 3) \in A \tag{12}$$

$$(1, 2, \text{“}Lillian'') \in A \tag{13}$$

Além disso, $\mathcal{SCRIPT}$ permite que sejam estendidos domínios de tuplas. O programa a seguir, por exemplo, define A como o domínio das tuplas que possuem dois inteiros e B como o domínio das tuplas que, além de possuírem dois campos inteiros, possuem também um campo com uma cadeia de caracteres:

```
DOMAINS
    A = (N, N)
    B = A EXT (Q)
```

Qualquer função que aceita um membro do domínio A como um parâmetro de entrada deve aceitar também um membro do domínio B, que constitui uma extensão de A:

```
DEFINITIONS
    DEF f(a:A) : A = a
...
    f(a) = a
    f(b) = b
```

A linguagem *Haskell* não possui este conceito de tuplas. Em *Haskell* uma tupla representa um conjunto de elementos no qual a ordem em que eles aparecem é importante e nada mais. Desse modo as Equações 12 e 13 não são verdadeiras nesta linguagem. *Haskell* também não possui capacidade de extensão de tuplas, ou de qualquer tipo de dados, como é o caso da linguagem $\mathcal{SCRIPT}$. A tradução de tuplas de $\mathcal{SCRIPT}$ para algum tipo de dados em *Haskell* não é tão trivial com a tradução de construções mais simples.

5.2 Funções Associadas a Domínios

A linguagem *SCRIPT* permite que funções sejam associadas a algum domínio de tuplas. O programa abaixo, por exemplo, define a função *push* como associada ao domínio de tuplas denominado *Stk*.

```
DEF Stk.push(elem) : Stk = ...
```

A aplicação de *push* deve sempre estar associada a uma variável ou objeto que pertença ao domínio *Stk*. Uma função que está associada a um domínio de tuplas pode ser redefinida e associada a qualquer extensão desse domínio. Neste caso, os domínios das funções em todas as redefinições devem ser equivalentes.

Redefinições de funções formam uma hierarquia que concorda com as relações de extensão dos domínios associados. A redefinição válida é a mais baixa na hierarquia. Note aqui uma grande semelhança com as linguagens orientadas a objetos, como por exemplo *Java* [11]. No programa a seguir, a função *f*, associada inicialmente ao domínio das tuplas *(N, N)* foi redefinida para o domínio das tuplas *(N, N, N)*. É importante que cada ocorrência da função *f* no corpo de um programa *SCRIPT* seja associada a implementação correta. Esta capacidade é denominada "Associação Dinâmica" e está presente em quase todas as linguagens que permitem programação orientada a objetos. Observe que a expressão `f^(x)` representa uma chamada à função *f* associada ao domínio *A*.

```
    DOMAINS
        A = (N, N);
        B = A EXT (N)

    DEFINITIONS
1)      DEF A.f(x) = ...
2)      DEF B.f(x) = ... f^(x) ...
3)      DEF b = (1, 2, 3)
4)      DEF a = b
5)      DEF foo ... = a.f(...)
```

Na seqüência de definições, logo acima, a chamada à função *f* associada ao nome de domínio *a*, na linha 5, deve invocar a função definida na linha 2, pois o nome *a*, neste caso, está associado a uma função do domínio *B* (*b = (1, 2, 3)*). O sistema responsável pela execução de um programa como aquele precisa ser poderoso o suficiente para verificar qual a implementação da função *f* que deverá ser invocada. Quando esta verificação é feita em tempo de execução, ela é denominada *Associação Dinâmica*.

O empecilho à tradução, neste caso, é o fato da linguagem *Haskell* padrão não possuir associação dinâmica. Isto não constitui uma deficiência da linguagem, pois todos os conflitos entre tipos e a ligação entre chamadas a funções e a implementação dessas funções são resolvidos de maneira estática, durante a compilação. Este fato, gera, contudo, uma dificuldade extra que precisa ser contornada no processo de tradução proposto neste trabalho.

6 Soluções Propostas para Problemas de Tradução

Conforme detalhado na Seção 5, o processo de tradução de $\mathcal{SCRIPT}$ para *Haskell* é dificultado pela existência de importantes diferenças conceituais entre estas duas linguagens. Dentre estas diferenças, as mais notáveis se referem à forma como tuplas são vistas em ambas linguagens e aos princípios de Orientação a Objetos existente em $\mathcal{SCRIPT}$ e ausente em *Haskell*.

Embora os programas gerados automaticamente tratem satisfatoriamente as diferenças entre tuplas observadas nas duas linguagens, a tradução dos aspectos relacionados a Orientação a Objetos não foi implementada. Soluções, entretanto, já foram propostas, e poderão ser implementadas em trabalhos futuros.

6.1 O Tipo Universal

A fim de possibilitar aos programas produzidos em *Haskell* atribuírem às tuplas a mesma semântica que é atribuída a elas pela linguagem $\mathcal{SCRIPT}$, decidiu-se que os interpretadores gerados automaticamente trabalhariam com dados de apenas um tipo. Assim, todos os dados manipulados por estes programas, sejam eles valores inteiros, booleanos, strings ou tipos compostos, como listas, tuplas ou funções, são elementos pertencentes a um mesmo tipo algébrico, denominado *Tipo Universal*.

Definir todos os dados presentes nos interpretadores gerados automaticamente como valores pertencentes a um único tipo foi uma medida muito drástica, no sentido de que bastaria que apenas as tuplas relacionadas via o mecanismo de extensão e herança tivessem o mesmo tipo. Isto permitiria que uma tupla estendida fosse aceita em qualquer situação onde uma tupla primitiva pudesse ser esperada, como é visto no código abaixo, escrito em $\mathcal{SCRIPT}$:

```
    DOMAINS
        A = (N, N)
        B = A EXT (Q)
    DEFINITIONS
1)      DEF b = (1, 2, "Lillian")
2)      DEF a = b
```

```
    DOMAINS
        A = (N, N)
        B = A EXT (Q)
    DEFINITIONS
1)      DEF f (b) : Q = LET (?, ?, q) = b IN q
2)      DEF a = (1, 2)
3)      DEF q1 = f(a)
```

Figura 4: Programa $\mathcal{SCRIPT}$ incorreto

Na linha 2 do trecho de código apresentado, um elemento do domínio *a* é definido como tendo o valor de um elemento do domínio *b*, o que, na linguagem $\mathcal{SCRIPT}$ é perfeitamente natural, uma vez que o domínio *a* é estendido pelo domínio *b*. Este tipo de associação, entretanto, não existe na linguagem *Haskell*, conforme mostrado na Seção 5. A solução encontrada foi, dessa forma, definir os domínios *A* e *B* como pertencentes a um mesmo tipo, quando da tradução para um programa escrito em *Haskell*. Tal programa teria, então, o seguinte aspecto:

```
data U = TUPLE_A (Int, Int) | TUPLE_B (Int, Int, String)

b :: U          -- b pertence a U
b = TUPLE_B (1, 2, "Lillian")
a :: U          -- a pertence a U
a = b
```

O código mostrado, quando executado por um interpretador *Haskell* não desencadeará nenhum erro de verificação de tipos, uma vez que tanto *a*, quanto *b*, naquele programa, são valores pertencentes ao mesmo tipo *U*.

Esta solução, no entanto, permite que uma tupla primitiva seja encontrada onde somente uma tupla derivada o poderia ser, o que faz com que programas como o visto na Figura 4 sejam considerados corretos durante a verificação de tipos realizada sobre o correspondente código *Haskell* gerado. O gerador de interpretadores não realiza este tipo de correção durante a análise dos programas que estão sendo traduzidos, logo, deve-se supor que **os programas escritos na linguagem $\mathcal{SCRIPT}$ fornecidos ao gerador estão corretos**. Caso isto não seja verdade, então poderá ocorrer erros em tempo de execução nos programas gerados automaticamente. Na Figura 5 pode-se ver o programa *Haskell* correspondente ao código $\mathcal{SCRIPT}$ mostrado na Figura 4. Embora este programa seja considerado correto durante a verificação de tipos, ele provoca um erro de execução quando tenta-se obter o terceiro campo de uma estrutura que possui apenas dois campos.

```
data U = TUPLE_A (Int, Int) | TUPLE_B (Int, Int, String)

f :: U -> String
f b = let (_, _, q) = b in q
a :: U
a = TUPLE_A (1, 2)
q :: String
q1 = f a
```

Figura 5: Programa *Haskell* incorreto

```
type N = Int
type T = Bool
type Q = String

data U = BUILTN N | BUILTQ Q | BUILTT T | LIST [U] | ...
```

Figura 6: Mínima Definição do Tipo Universal

Embora bastasse que apenas famílias de tuplas relacionadas fossem definidas como pertencentes ao mesmo tipo, decidiu-se que todos os valores manipulados nos programas gerados automaticamente fossem elementos de um mesmo tipo, no caso o Tipo Universal. A principal razão por parte desta decisão foi facilitar o processo de geração de código traduzido. Como quaisquer valores presentes no código gerado pertencem ao mesmo tipo, não é necessário gerar diferentes tipos para diferentes famílias de tuplas. Todas as tuplas, tenham elas alguma relação ou não, passam a pertencer a um mesmo tipo. Dessa forma, todas as tuplas com, por exemplo, quatro campos, passam a ser representadas por um elemento do tipo universal que denota uma tupla de quatro campos.

Os tipos primitivos passaram a fazer parte, portanto, do Tipo Universal, de maneira que não mais torna-se necessário conhecer o tipo de cada campo de uma determinada tupla. Não importa que valor seja atribuído àquele campo, pois qualquer que seja ele, será um elemento do tipo universal.

O Tipo Universal é criado de acordo com os tipos encontrados no código $\mathcal{SCRIPT}$ que está sendo traduzido. Alguns valores do Tipo Universal, entretanto, são pré-definidos. Este é o caso dos tipos primitivos (valores inteiros, booleanos e cadeias de caracteres) e também das listas. Assim, todo interpretador gerado automaticamente possui necessariamente a definição de *data U* mostrada na Figura 6, além de outras definições, de acordo com o programa que foi traduzido.

O Tipo Universal é representado em *Haskell* como um tipo algébrico. Tipos

algébricos permitem que sejam definidos tipos recursivos, como por exemplo listas e árvores e valem-se de entidades denominadas *construtores* para que diferentes valores possam ser tratados de forma diferente por uma mesma função. O tipo *data U* (Tipo Universal) presente nos programas gerados automaticamente apresenta os seguintes tipos de construtores:

BUILTN Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como um número inteiro, ou seja, um membro do domínio N.

BUILTQ Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como uma cadeia de caracteres, ou seja, um membro do domínio Q.

BUILTT Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como um valor booleano, ou seja, como um membro do domínio dos valores booleanos (T).

LIST Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como uma lista. Cada um dos elementos que compõem esta lista também pertencem ao Tipo Universal.

NODE Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como um nodo. Este tipo de estrutura é descrita com mais detalhes na Seção 6.3.

TUPLE Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como uma tupla de dados. Cada um dos elementos desta tupla pertence também ao Tipo Universal.

FUNC Este construtor indica que o valor representado pelo Tipo Universal deve ser tratado como uma função.

6.2 Estratégias para Manipulação de Tuplas

Os mecanismos de extensão e a hierarquia presentes entre as tuplas da linguagem $\mathcal{SCRIPT}$ constituíram uma das principais dificuldades para a tradução desta linguagem para *Haskell*. Tais dificuldades levaram à adoção de uma série de procedimentos de tradução que serão mostrados nesta subseção.

Representação de tuplas As tuplas, nos programas traduzidos, são representadas como elementos do Tipo Universal. A representação de tuplas, no Tipo Universal, difere com relação ao número de campos. Assim, existe um construtor para representar tuplas com apenas um campo, com dois campos, e com quantos campos for necessário. Caso, por exemplo, a maior tupla encontrada em um

programa que está sendo traduzido possua cinco campos, então será produzida a seguinte definição para o Tipo Universal:

```
data U = ... TUPLE0 () | TUPLE1 (U) | TUPLE2 (U, U) | TUPLE3 (U, U, U) |
        TUPLE4 (U, U, U, U) | TUPLE5 (U, U, U, U, U) ...
```

Extração de campo de tupla A extração de um campo de tupla é feita pela função *selectField(tup, n)*, que, dado uma tupla (*tup*), devolve seu *n-ésimo* campo. Tal função é definida de modo a estar apta a lidar com a maior tupla que pode ser encontrada no programa $\mathcal{SCRIPT}$ original. Caso a maior tupla encontrada em termos do seu número de campos fosse, por exemplo, uma tupla com três elementos, seria gerada a seguinte função *selectField*:

```
selectField :: U -> Int -> U
selectField (TUPLE1 (c0)) 1 = c0
selectField (TUPLE2 (c0, c1)) 1 = c0
selectField (TUPLE2 (c0, c1)) 2 = c1
selectField (TUPLE3 (c0, c1, c2)) 1 = c0
selectField (TUPLE3 (c0, c1, c2)) 2 = c1
selectField (TUPLE3 (c0, c1, c2)) 3 = c2
```

Associação de Tuplas a Nomes de Domínios No corpo de uma definição de função é possível associar uma seqüência de nomes de campos a um nome de domínio, via a cláusula `LET`, como, por exemplo, no trecho de código abaixo:

```
DEF f alfa : N = LET (n1, n2, n3) = alfa IN n1 PLUS n2 PLUS n3
```

Neste trecho de código, o parâmetro *alfa* denota uma tupla que possui os primeiros três campos do tipo inteiro. Espera-se, então, que a função *f* aceite um parâmetro que denote uma tupla com no mínimo três campos, podendo, contudo, estar este parâmetro associado a uma tupla com um número maior de campos. Para que tal construção possa ser corretamente traduzida para *Haskell*, é preciso que os campos relevantes da tupla *alfa* possam ser extraídos no momento da associação. Isto é feito pela função *extract(tup, n)*, cuja definição pode ser vista logo a seguir:

```
extract :: U -> Int -> U
extract (TUPLE0 ()) 0 = TUPLE0 ()
extract (TUPLE1 (c0)) 0 = TUPLE0 ()
extract (TUPLE1 (c0)) 1 = TUPLE1 (c0)
extract (TUPLE2 (c0, c1)) 0 = TUPLE0 ()
```

```
extract (TUPLE2 (c0, c1)) 1 = TUPLE1 (c0)
extract (TUPLE2 (c0, c1)) 2 = TUPLE2 (c0, c1)
extract (TUPLE3 (c0, c1, c2)) 0 = TUPLE0 ()
extract (TUPLE3 (c0, c1, c2)) 1 = TUPLE1 (c0)
extract (TUPLE3 (c0, c1, c2)) 2 = TUPLE2 (c0, c1)
extract (TUPLE3 (c0, c1, c2)) 3 = TUPLE3 (c0, c1, c2)
```

Assim como a função *selectField*, esta função também é definida no momento da tradução tendo como base a tupla com o maior número de campos encontrada no código $\mathcal{SCRIPT}$ sob análise. O código logo acima, por exemplo, contém a definição da função *extract* para um $\mathcal{SCRIPT}$ cuja maior tupla possuía apenas três campos. A tradução para a chamada à função *f* com o parâmetro *alfa*, mostrada logo acima, pode ser vista no trecho de código a seguir:

```
f alfa = let (TUPLE3 (n1, n2, n3)) = extract alfa 3 in n1 + n2 + n3
```

Atualização de campos de tuplas Uma nova tupla pode ser criada a partir de uma outra redefinindo-se alguns dos campos da tupla original. Na linguagem $\mathcal{SCRIPT}$ este tipo de construção tem a forma: $t\{f_1 = v_1, \ldots, f_n = v_n\}$, onde t e v_i são expressões e f_i é um nome de campo, para $1 \leq i \leq n$. Esta operação cria uma nova tupla que tem os mesmos componentes da tupla resultante da avaliação da expressão t, exceto que os componentes identificados pelos campos f_i contêm os valores v_i.

Como um nome de tupla $\mathcal{SCRIPT}$ pode estar associado a tuplas com diferentes números de campos, a fim de traduzir este tipo de construção para *Haskell*, foi necessário que cada possível tipo da tupla em questão fosse explicitado. Por exemplo, a Figura 7 contém a tradução do trecho de código logo abaixo:

```
DOMAINS
    A = (N);
    B = A EXT (N);
    C = B EXT (N)

DEFINITIONS
    DEF atualiza a n : A = a{x=3}
```

Concatenação de tuplas A linguagem $\mathcal{SCRIPT}$ também permite que novas tuplas sejam criadas por meio do operador binário `CAT`, o qual concatena duas tuplas para construir uma maior. Novamente, como um nome de tupla pode denotar uma estrutura com um número de campos variável, foram necessárias algumas

```
atualiza a n = case a of
    TUPLE1 (campo0) -> TUPLE1 ((BUILTN 3))
    TUPLE2 (campo0, campo1) -> TUPLE2 ((BUILTN 3), campo1)
    TUPLE3 (campo0, campo1, campo2) -> TUPLE2 ((BUILTN 3), campo1, campo2)
```

Figura 7: Atualização de tuplas

adaptações neste operador para que tal construção pudesse ser traduzida para *Haskell*. A função `CAT` passou a tratar em separado cada possível associação entre seus parâmetros reais e formais. Abaixo encontra-se uma definição deste operador gerada quanto da tradução de um arquivo no qual a maior tupla encontrada possuía três campos:

```
cat :: U -> U -> U
cat (TUPLE0 ()) (TUPLE0 ()) = (TUPLE0 ())
cat (TUPLE0 ()) (TUPLE1 (c0)) = (TUPLE1 (c0))
cat (TUPLE0 ()) (TUPLE2 (c0, c1)) = (TUPLE2 (c0, c1))
cat (TUPLE0 ()) (TUPLE3 (c0, c1, c2)) = (TUPLE3 (c0, c1, c2))
cat (TUPLE1 (c0)) (TUPLE0 ()) = (TUPLE1 (c0))
cat (TUPLE1 (c0)) (TUPLE1 (c1)) = (TUPLE2 (c0, c1))
cat (TUPLE1 (c0)) (TUPLE2 (c1, c2)) = (TUPLE3 (c0, c1, c2))
cat (TUPLE2 (c0, c1)) (TUPLE0 ()) = (TUPLE2 (c0, c1))
cat (TUPLE2 (c0, c1)) (TUPLE1 (c2)) = (TUPLE3 (c0, c1, c2))
cat (TUPLE3 (c0, c1, c2)) (TUPLE0 ()) = (TUPLE3 (c0, c1, c2))
```

6.3 Estratégias para Tradução de Funções

Conforme fora dito na Sessão 6.1 deste relatório, todos os valores manipulados nos programas traduzidos para *Haskell* pertencem a um único tipo, denominado Tipo Universal. Também as funções foram definidas como elementos pertencentes ao tipo universal, porém apenas as funções cujas definições foram declarada no módulo `DOMAINS` de um programa $\mathcal{SCRIPT}$ foram tratadas desta forma. As funções apenas definidas no módulo `DEFINITIONS` destes programas não foram incluídas como elementos do tipo universal. A fim de incluir as funções como elementos do Tipo Universal, foi definido um novo construtor, denominado `FUNC` para aquele tipo algébrico. As definições de função encontradas no módulo `DOMAINS` seriam adicionadas como nomes de campos deste construtor. Na Figura 8 encontra-se a tradução do trecho de código a seguir:

```
DOMAINS
    F   = N -> N;
```

```
data U = FUNC{F_func :: U -> U, G_func :: U -> U -> U, H_func :: U -> U -> U}
```

Figura 8: Inclusão de funções ao Tipo Universal

```
data U = FUNC{F_func :: U->U}

f = FUNC{F_func = \n -> n + 1}
```

Figura 9: Declaração de funções pertencentes ao Tipo Universal

```
    G   = N -> N -> N;
    H   = Q -> N -> Q
```

Declaração de Função O módulo `DOMAINS` de um programa $\mathcal{SCRIPT}$ contém apenas definições de domínios de funções, ou, em outras palavras, seu protótipo. Funções definidas naquele módulo precisam ser efetivamente declaradas e implementadas, antes de serem utilizadas, o que apenas acontece no módulo `DEFINITIONS` de algum $\mathcal{SCRIPT}$. Dessa forma, durante a tradução de declarações de funções, para cada função, se ela houver sido declarada como um domínio, deve-se atualizar o seu identificador de campo no construtor `FUNC` de *data U*.

```
DOMAINS
    F   = N -> N
DEFINITIONS
    f n : N = n PLUS 1
```

A tradução do trecho de código acima pode ser vista na Figura 9. Construtores, na linguagem *Haskell* podem ter campos identificados por certos nomes, o que permite que tais campos sejam posteriormente atualizados, como é este o caso.

Chamada de Função Ao inserir-se as funções declaradas no módulo `DOMAINS` no Tipo Universal, os programas traduzidos para *Haskell* passaram a ter dois tipos de funções: as que pertencem a *data U* e as que não pertencem. Assim, para traduzir chamadas de funções, tornou-se necessário verificar se a função alvo foi declarada no módulo `DOMAINS`. Caso isto tenha sido feito, então o identificador de campo atribuído a esta função no momento de sua inclusão ao Tipo Universal deve ser invocado quando de sua chamada, como no trecho de código abaixo, no qual a função *f*, cuja declaração pode ser vista na Figura 9 é chamada:

```
n = F_func f 3
```

Atualização de Função É bastante comum, em Semântica Denotacional, que funções sejam definidas em um padrão progressivo. Por exemplo, inicialmente uma função é definida de modo a retornar o valor indefinido ("?") para qualquer de seus parâmetros de entrada. Posteriormente, os valores de retorno desta função são gradualmente modificados para determinados parâmetros de entrada. O termo "atualização de função" é apenas um abuso de linguagem, pois uma nova função é produzida, de acordo com o paradigma funcional.

Funções que modelam a memória de um dispositivo computacional, por exemplo, são alvo de constantes atualizações, sempre que uma nova variável é declarada e preenchida com um novo valor, como se uma posição desta memória simulada passasse a possuir um conteúdo válido. O trecho de código abaixo denota a declaração de uma função deste modo e a declaração de uma variável, denominada *beta*, que passa a armazenar o valor inteiro *1*:

```
DOMAINS
    S   = Q -> N

DEFINITIONS
    DEF s q = "unused"
    DEF s1 q = s{beta = 1}q
```

A tradução da atualização de uma função inclusa ao Tipo Universal é feita redefinindo-se a função denotada pelo identificador de campo atribuído àquela função. Por exemplo, para o código visto logo acima, ter-se-ia a tradução mostrada a seguir:

```
data U = FUNC {F_func :: U -> U}

s = FUNC{F_func = \q -> (BUILTQ "unused")}

s1 = FUNC {F_func = \q ->   if q == (BUILTQ "beta") then (BUILTN 1)
                            else F_func s q}
```

6.4 Análise das Soluções Propostas

Ao incluir quase todos os dados presentes nos programas traduzidos ao Tipo Universal, apenas quatro tipos de valores passaram a ser encontrados nos códigos gerados automaticamente: o próprio Tipo Universal, as listas compostas por elementos do Tipo Universal, as funções que não foram declaradas no módulo `DOMAINS` de um $\mathcal{SCRIPT}$ e os valores booleanos, presentes em estruturas condicionais.

É provável que uma solução mais simples e elegante pudesse ser encontrada, criando-se mais de um "Tipo Universal": um tipo algébrico para cada família de

tuplas. Neste caso, seria necessário que as relações entre tuplas fossem convenientemente armazenadas em tabelas de símbolos, de modo que grupos de tuplas equivalentes pudessem ser convenientemente manipulados. Também o tipo de cada função deveria ser armazenado durante a análise de código a ser traduzido, a fim de que o polimorfismo fosse corretamente implementado.

A extensão do Tipo Universal à maior parte dos dados presentes nos programas traduzidos facilitou a geração de código principalmente no que toca o tratamento de tuplas, uma vez que não foi necessário armazenar o tipo dos campos destas estruturas em tabela de símbolos. Diversas dificuldades, entretanto, surgiram devido à solução adotada: passou a haver dois tipos diferentes de funções, tratadas de maneira diferente; listas compostas por elementos do Tipo Universal, e valores booleanos presentes em testes de estruturas condicionais exigiram maior cuidado no momento da tradução, uma vez que não são elementos que fazem parte do Tipo Universal. Também foi preciso definir diversas operações adequadas aos elementos presentes no Tipo Universal, o que contribuiu para deixar os programas gerados automaticamente bastante extensos em termos de linhas de código.

7 Geração de Interpretadores e Interpretação de Programas

Esta Seção visa expor ao leitor uma visão geral a respeito do ambiente de programação criado e de todo o processo envolvido na geração automática de interpretadores e na interpretação de programas. A fim de melhorar a legibilidade deste relatório, será utilizada a seguinte lista de símbolos:

$\mathcal{L}$: uma linguagem qualquer, pertencente a qualquer paradigma de programação: imperativo, funcional ou lógico. Por exemplo: *Haskell* [7] ou *C* [13].

$\mathcal{S}_{\mathcal{L}}$: o código fonte de um programa $\mathcal{SCRIPT}$ que contém a definição semântica de uma linguagem $\mathcal{L}$ qualquer. Pretende-se que $\mathcal{S}_{\mathcal{L}}$ contenha a descrição formal de cada um dos comandos, expressões e declarações daquela linguagem.

P_L : o código fonte de um programa escrito em uma linguagem $\mathcal{L}$.

$\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$: uma representação da árvore de sintaxe abstrata que descreve a estrutura de P_L. Esta árvore pode ser obtida a partir da análise sintática de P_L. Maiores referências sobre esta estrutura podem ser encontradas na Subseção 7.2, neste manual.

$\mathcal{T}_{\mathcal{S}\rightarrow\mathcal{H}}$: um programa capaz de gerar um interpretador para $\mathcal{L}$ a partir de $\mathcal{S}_{\mathcal{L}}$.

$\mathcal{I}_{\mathcal{L}}$: interpretador de $\mathcal{L}$, gerado automaticamente por $\mathcal{T}_{\mathcal{S}\to\mathcal{H}}$. $\mathcal{I}_{\mathcal{L}}$ simula a execução de um programa P_L quando alimentado com $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$.

τ_L : tabela de símbolos produzida por $\mathcal{T}_{\mathcal{S}\to\mathcal{H}}$. Esta tabela contém todos os símbolos encontrados na definição dos domínios de $\mathcal{S}_{\mathcal{L}}$.

$\mathcal{T}_{\mathcal{P}\to\mathcal{A}}$: um programa capaz de gerar tradutores para $\mathcal{L}$ a partir de $\mathcal{S}_{\mathcal{L}}$ e de τ_L. Tais tradutores convertem P_L para $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$. $\mathcal{T}_{\mathcal{P}\to\mathcal{A}}$ e $\mathcal{T}_{\mathcal{S}\to\mathcal{H}}$ constituem o produto final desenvolvido como resultado deste Projeto Orientado em Computação.

$\mathcal{T}_{\mathcal{P}_{\mathcal{L}}\to\mathcal{A}_{\mathcal{P}}}$: um programa tradutor. Este programa, que é gerado automaticamente por $\mathcal{T}_{\mathcal{P}\to\mathcal{A}}$ traduz programas escritos em $\mathcal{L}$ para uma estrutura de árvore sintática que os representa: $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$.

$P_L(In)$: o resultado da execução de P_L sobre a entrada de dados *In*.

O processo no qual um interpretador para $\mathcal{L}$ é gerado automaticamente e programas P_L são interpretados é longo, e portanto sua descrição será divida em quatro partes. Esta complexidade se explica em parte pelo fato de os interpretadores gerados automaticamente não receberem diretamente como entrada o código fonte dos programas que deverão ser interpretados. As partes que compõem tal processo podem ser resumidas da seguinte forma:

1. Na primeira parte $\mathcal{T}_{\mathcal{S}\to\mathcal{H}}$ é utilizado para produzir $\mathcal{I}_{\mathcal{L}}$ e τ_L a partir de $\mathcal{S}_{\mathcal{L}}$.

2. A segunda parte do processo consiste na geração de $\mathcal{T}_{\mathcal{P}_{\mathcal{L}}\to\mathcal{A}_{\mathcal{P}}}$. A partir de $\mathcal{S}_{\mathcal{L}}$ e de τ_L, $\mathcal{T}_{\mathcal{P}\to\mathcal{A}}$ produz um tradutor capaz de converter P_L para $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$.

3. A terceira parte do processo compreende a tradução de programas. O interpretador ($\mathcal{I}_{\mathcal{L}}$) produzido na etapa anterior não recebe diretamente o código fonte dos programas que deverão ser interpretados (P_L). É necessário, portanto, a tradução de P_L para um formato adequado, neste caso, $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$.

4. Na quarta e última parte deste processo acontece a interpretação de programas. $\mathcal{I}_{\mathcal{L}}$ é executado, tendo como entrada $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$. O resultado da execução de $\mathcal{I}_{\mathcal{L}}$ sobre $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$ é $P_L(In)$, ou seja, a saída que seria produzida por uma versão compilada de P_L quando alimentado com os dados de entrada aqui denominados *In*.

Na Figura 10 encontra-se representado o processo de geração automática de interpretadores e a interpretação de programas. Nesta figura, as quatro fases que compõem todo o processo encontram-se aglutinadas em um único esquema. Cada uma das etapas anteriormente apresentadas será descrita com maiores detalhes

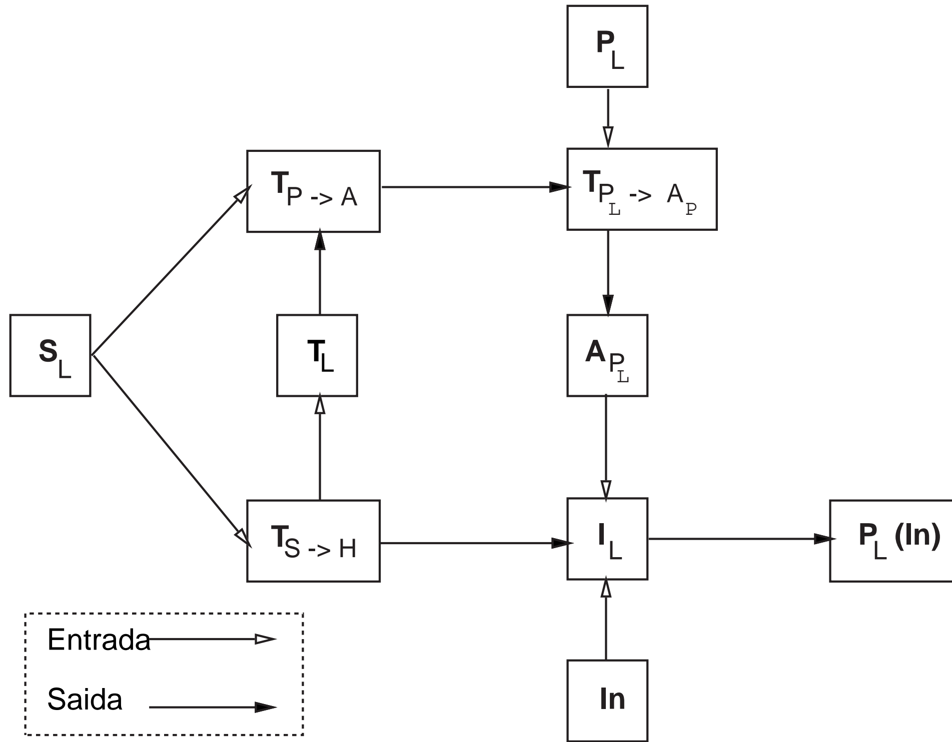

Figura 10: Geração automática de interpretadores e interpretação

nas Subsessões 7.1, 7.2, 7.3 e 7.4. A fim de melhor demonstrar como interpretadores podem ser criados e efetivamente utilizados, será mostrado, ao longo destas subsessões, a interpretação de programas escritos na linguagem *Arit*, definida na Seção 3.

7.1 Primeira Etapa – Geração de Interpretadores

Nesta primeira etapa, a ferramenta $\mathcal{T}_{\mathcal{S}\rightarrow\mathcal{H}}$ recebe como entrada a descrição de uma certa linguagem $\mathcal{L}$ contida em $\mathcal{S}_{\mathcal{L}}$ e a partir dela são produzidos um interpretador para $\mathcal{L}$, escrito em *Haskell* e também uma tabela de símbolos (τ_L). A interação entre tais elementos pode ser vista na Figura 11.

$\mathcal{I}_{\mathcal{L}}$ é um programa escrito na linguagem *Haskell*. Este programa é composto por dois arquivos fonte. Embora nomes neste caso não sejam de muita relevância, uma vez que o usuário do sistema tem liberdade para defini-los como desejar, neste relatório tais arquivos serão chamados `Main.hs` e `Universal.hs`. O primeiro arquivo contém a tradução para a linguagem *Haskell* das definições presentes no módulo `DEFINITIONS` de $\mathcal{S}_{\mathcal{L}}$. O código gerado para o $\mathcal{SCRIPT}$ visto na Figura 3

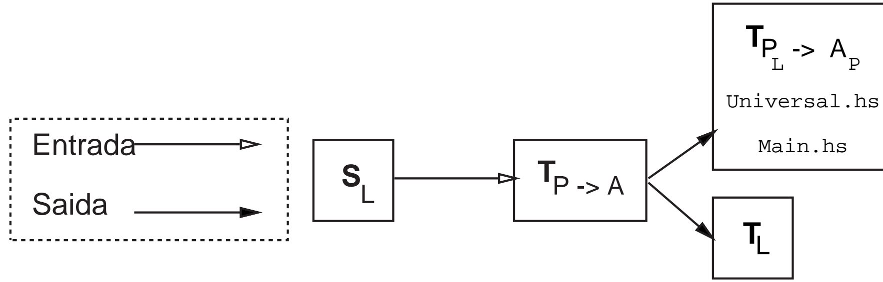

Figura 11: A geração de interpretadores

```
module Main where
import Program (command)
import Universal

elabExp TUPLE1 (exp) =
    case exp of
        (NODE5 exp1 (BUILTQ "+") exp2) -> let
            n1 = elabExp exp1;
            n2 = elabExp exp2 in n1 + n2
        (NODE6 n) -> n
```

Figura 12: Arquivo `Main.hs`

pode ser visto na Figura 12.

O segundo arquivo fonte, aqui chamado `Universal.hs` contém as definições dos tipos criados a partir da análise do módulo `DOMAINS` de $\mathcal{S}_{\mathcal{L}}$. Conforme visto na Seção 5, os interpretadores gerados automaticamente possuem somente um tipo de dados, denominado *Universal*. Além destas definições, o arquivo `Universal.hs` contém também a definição de diversas funções necessárias à manipulação dos elementos que compõem o Tipo Universal. A Figura 13 mostra o conteúdo do arquivo `Universal.hs` criado a partir do módulo `DOMAINS` presente no $\mathcal{SCRIPT}$ visto na Figura 3. Dado o tamanho do código, as funções, classes e instâncias auxiliares não são mostradas.

Além de produzir $\mathcal{I}_{\mathcal{L}}$, $\mathcal{T}_{\mathcal{S}\to\mathcal{H}}$ também gera um arquivo de texto (ASCII), contendo os símbolos encontrados nas definições de domínios de $\mathcal{L}$. Tal arquivo é utilizado na próxima etapa deste processo, como entrada para $\mathcal{T}_{\mathcal{P}\to\mathcal{A}}$, conforme será mostrado na Subsessão 7.2 deste relatório. O arquivo correspondente à linguagem *Arit* pode ser visto na Figura 14. Neste arquivo, cada linha descreve a estrutura de um domínio utilizado na definição de $\mathcal{L}$. Os números que finalizam

```
module Universal where

type N = Int
type T = Bool
type Q = String

data U = BUILTN N | BUILTQ Q | BUILTT T | LIST [U] | NODE6 U | NODE5 U U U |
         TUPLE0 () | FUNC{}

class ...

instance ...
```

Figura 13: Arquivo `Universal.hs`

```
[Exp "+" Exp]|5
[N]|6
```

Figura 14: Tabela de símbolos gerada para *Arit*

cada linha indicam quais os nodos correspondentes a cada domínio, ou seja, ao domínio $Exp = Exp + Exp$ corresponde o nodo de número 5 (`NODE5`), no arquivo `Universal.hs`.

7.2 Segunda Parte – Geração de Tradutores

Os interpretadores produzidos automaticamente, conforme descrito na Subseção 7.1 não recebem como entrada o código fonte dos programas que deverão ser interpretados, ao contrário de grande parte dos interpretadores largamente utilizados. Entre outras razões, isto se deve às dificuldades inerentes à linguagem *Haskell* para efetuar leitura e escrita de dados em arquivos binários. Assim, antes de serem fornecidos para $\mathcal{I}_{\mathcal{L}}$ os programas escritos em $\mathcal{L}$ precisam ser traduzidos para um formato adequado.

A tradução de P_L para $\mathcal{A}_{\mathcal{P}_{\mathcal{L}}}$ é feita por $\mathcal{T}_{\mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{A}_{\mathcal{P}}}$. Esta última ferramenta é gerada automaticamente por $\mathcal{T}_{\mathcal{P} \rightarrow \mathcal{A}}$ a partir dos módulos `LEXIS` e `SINTAXE` de $\mathcal{S}_{\mathcal{L}}$. A geração de $\mathcal{T}_{\mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{A}_{\mathcal{P}}}$ acontece na segunda etapa do processo de geração de interpretadores e interpretação, que é, dentre as quatro fases do processo, a mais complexa.

$\mathcal{T}_{\mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{A}_{\mathcal{P}}}$ é composto por um analisador sintático, um analisador léxico e rotinas auxiliares. O analisador léxico é um programa escrito de acordo com a notação utilizada pela ferramenta *lex* [12] e é produzido a partir da tradução do bloco

```
/* Regular Definitions */
digit           ([0-9])
%%
"+"             { yylval.quote = strdup(yytext); return(TOKEN_1); }
{digit}+        { yylval.quote = strdup(yytext); return NUM_TK; }
\n              { lineno++; }
.               ;
%%
```

Figura 15: Analisador léxico gerado para a linguagem *Arit*

`LEXIS` de $\mathcal{S}_{\mathcal{L}}$. Para o $\mathcal{SCRIPT}$ visto na Figura 1 seria gerado o analisador léxico mostrado na Figura 15. Tal código é escrito em um arquivo denominado `lex.l`, de maneira que este possa ser, posteriormente, fornecido à ferramenta `Lex` para que um programa final possa ser gerado.

O analisador sintático, por sua vez, é um programa escrito segundo a notação reconhecida pela ferramenta `Yacc` [12], e que faz uso de rotinas auxiliares para criar uma árvore de sintaxe com os componentes gramaticais dos programas que estiverem sendo analisados. O analisador sintático correspondente ao $\mathcal{SCRIPT}$ mostrado na Figura 1 pode ser visto na Figura 16. Este código é gravado em um arquivo denominado `yacc.y`, de maneira que possa ser fornecido como entrada para a ferramenta `yacc`, que a partir dele irá gerar o *parser* propriamente dito. Além da gramática de uma linguagem de programação, o arquivo `yacc.y` contém uma série de rotinas auxiliares para a construção da árvore traduzida. Tais rotinas são escritas na linguagem C [13] e não foram mostradas por questões de espaço.

Além dos analisadores léxico e sintático gerados nesta etapa, são produzidos também dois outros arquivos: `header.h` e `syntax.make`. O primeiro deles contém a definição das funções e tipos auxiliares utilizados para a construção da árvore de sintaxe dos programas que estão sendo traduzidos. O segundo, por sua vez, contém as diretivas de compilação de todos os programas gerados nesta fase. Na Figura 17 estão representadas as iterações entre os elementos presentes nesta fase do processo.

7.3 Terceira Parte – Geração de Árvore de Sintaxe

$\mathcal{I}_{\mathcal{L}}$ recebe como entrada a árvore de sintaxe que descreve o programa a ser interpretado. Estas árvores são representadas em *Haskell* como um tipo de dados recursivo, que faz parte do conjunto de tipos que compõe o Tipo Universal. Por exemplo, $1 + 2 + 3$ é uma expressão que denota uma soma de três números inteiros e pode ser vista como um programa escrito na linguagem *Arit*. Para tal programa

```
%%
aux_1   : exp                   {gera_saida($1);}
exp     : exp TOKEN_1 exp       {$$ = gera_arv($1, cria_folha($2, NULL), NULL);
                                 $$ = gera_arv($$, $3, "Exp");}
        | num                   {$$ = gera_arv($1, NULL, "Exp");}
num     : NUM_TK                {$$ = cria_folha($1, "N");}
%%
int main(int argc, char **argv);
int binSearch (char *ch, domCel *t, int cap);
void iniciaTabela (domTable *tab);
char *percorre (arvore a, char** s);
arvore gera_arv (arvore a1, arvore a2, char *label);
arvore cria_folha (char *s, char *t);
void gera_saida (arvore a);
```

Figura 16: Analisador léxico gerado para a linguagem *Arit*

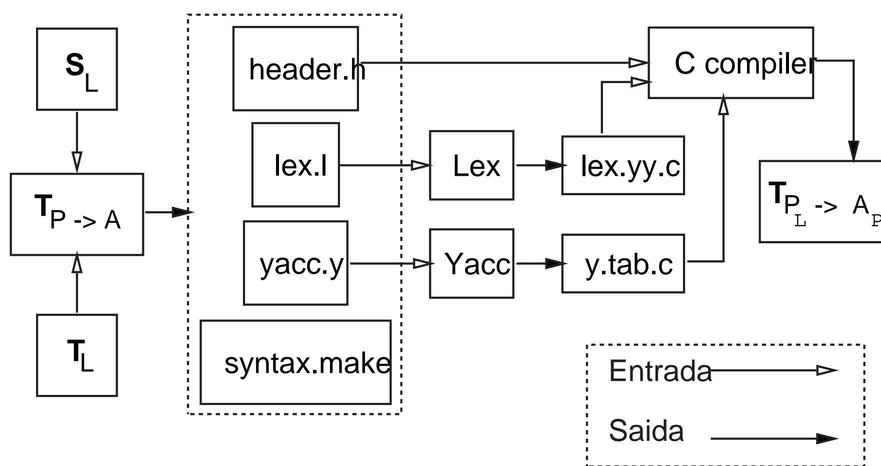

Figura 17: Geração de tradutores

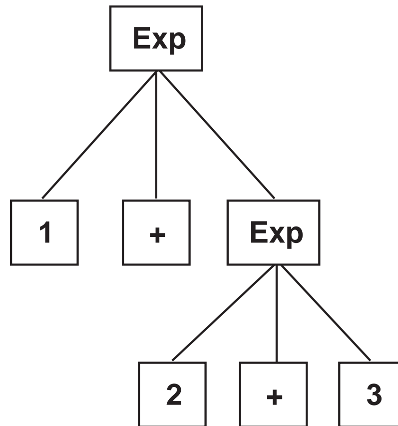

Figura 18: Árvore sintática que representa o programa *1 + 2 + 3*

```
module Program (command) where
import Universal

command :: U
command = (NODE5 (NODE6 (BUILTN 1) ) (BUILTQ "+") (NODE5 (NODE6 (BUILTN 2) )
(BUILTQ "+") (NODE6 (BUILTN 3) )))
```

Figura 19: módulo `Program` que contém uma representação da árvore de sintaxe para o programa *1 + 2 + 3*

seria gerada a árvore sintática vista na Figura 18:

Esta árvore sintática é fornecida para $\mathcal{I}_{\mathcal{L}}$ como um módulo separado da linguagem *Haskell*. $\mathcal{T}_{\mathcal{P}_{\mathcal{L}} \rightarrow \mathcal{A}_{\mathcal{P}}}$ gera, a partir de P_L, um arquivo texto contendo o código de um módulo denominado `Program`. Este módulo, por sua vez, importa a definição de Tipo Univesal do arquivo `Universal.hs` e exporta uma única definição, denominada `command`. Esta representa a árvore sintática que descreve P_L. Tal definição é importada por $\mathcal{I}_{\mathcal{L}}$, conforme pode ser visto na Figura 12. No módulo `Program`, a árvore vista na Figura 18 é representada como um valor pertencente ao Tipo Universal. Embora o sistema permita ao usuário a liberdade para definir o nome do arquivo em que ficará armazenado o módulo `Program`, alguns interpretadores *Haskell* exigem que este seja denominado `Program.hs`. Na Figura 19 pode ser visto o conteúdo do arquivo `Program.hs` que representa a árvore mostrada na Figura 18.

O código visto na Figura 19 é produzido pelo tradutor gerado automaticamente na etapa anterior. A interação entre os elementos presentes nesta fase do processo está representada na Figura 20. Esta é a mais simples das quatro fases de todo

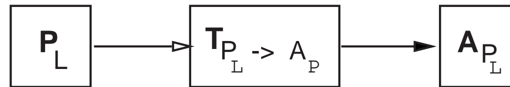

Figura 20: Tradução de programas para formato de árvore sintática

o processo de geração de interpretadores e interpretação, pois envolve somente a execução de $\mathcal{T}_{\mathcal{P}_\mathcal{L} \rightarrow \mathcal{A}_\mathcal{P}}$ sobre a entrada apropriada, no caso os programas que deverão ser traduzidos (P_L).

7.4 Quarta Parte – Interpretação de Programas

Na quarta e última etapa do processo dá-se a interpretação de programas, propriamente dita. Estes, tendo sido já convertidos para um formato apropriado, são fornecidos ao interpretador produzido durante a primeira fase.

Para a realização desta fase, é necessário um ambiente no qual programas escritos na linguagem *Haskell* possam ser executados. O mais natural, portanto, é a utilização de algum interpretador desta linguagem, como por exemplo, o interpretador `HUGS` [8]. A associação entre os três arquivos necessários fica a cargo do interpretador *Haskell*, dessa forma.

É importante que fique claro que a árvore sintática que descreve o programa que será interpretado não é fornecida como entrada ao interpretador, no sentido de que este deve "lê-la" a partir de algum dispositivo de fluxo de dados. Na realidade, tal árvore de sintaxe está contida em um programa da linguagem *Haskell*, que fica disponível à $\mathcal{I}_\mathcal{L}$ via o mecanismo de importação de definições da própria linguagem *Haskell*. Isto seria como se o interpretador ($\mathcal{I}_\mathcal{L}$) e o arquivo a ser interpretado ($\mathcal{A}_{\mathcal{P}_\mathcal{L}}$) fossem compilados juntos, no momento da execução. Esta etapa final encontra-se esquematizada na Figura 21.

8 Princípios de Funcionamento dos Interpretadores Gerados

Nesta Seção procurar-se-á explicar os princípios que regem o funcionamento dos interpretadores gerados automaticamente a partir da descrição semântica de uma linguagem de programação.

Um programa escrito em uma linguagem de programação $\mathcal{L}$ pode ser entendido como uma seqüência de comandos, os quais, por sua vez, são formados por expressões, outros comandos e *palavras primitivas*, como por exemplo *if* e *while*. Tais comandos e expressões podem ser organizados de acordo com a sua disposição

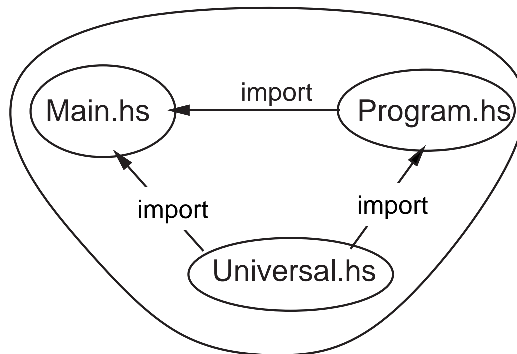

Figura 21: Relação entre os arquivos *Haskell* gerados

sintática no programa, em uma estrutura denominada árvore de sintaxe, descrita na Seção 7.

Conforme foi visto na Seção 2, uma completa descrição semântica de uma linguagem de programação $\mathcal{L}$ fornece de forma não ambígua o exato significado de cada construção típica daquela linguagem. A partir da descrição semântica de um comando ou expressão é possível definir uma série de rotinas cuja execução eqüivale à execução da construção em análise.

O processo de interpretação consiste em percorrer a árvore de sintaxe de um programa, executando as rotinas adequadas para cada construção completamente reconhecida. Considerando a linguagem *Arit*, definida na Seção 3, por exemplo, existem dois tipos de construções que podem ser completamente reconhecidas ao ser percorrida a árvore de sintaxe de algum programa nela codificado: números inteiros, ou somas de duas expressões. No primeiro caso, o interpretador deve retornar o valor do número encontrado, ao passo que no segundo, o interpretador deve avaliar separadamente cada subexpressão e retornar a sua soma. A Figura 22 mostra a seqüência de reduções realizada por um interpretador ao ser percorrida a árvore gerada para o programa `1 + 2 + 3`.

A fim de representar a árvore de sintaxe de programas, a linguagem $\mathcal{SCRIPT}$ faz uso de uma estrutura denominada *nodo*. Domínios de nodos têm a forma $[D_1 \ldots D_n]$, onde cada D_i, $1 \leq i \leq n$ pode denotar ou uma cadeia de caracteres ou um identificador de domínio. Sendo esta uma estrutura recursiva, presta-se bem para representar árvores de dados. A árvore vista na Figura 18, por exemplo, seria representada em um $\mathcal{SCRIPT}$ da maneira descrita no trecho de código a seguir:

```
[[1] "+" [[2] "+" [3]]]
```

Um valor como o descrito acima pode ser fornecido como entrada para um

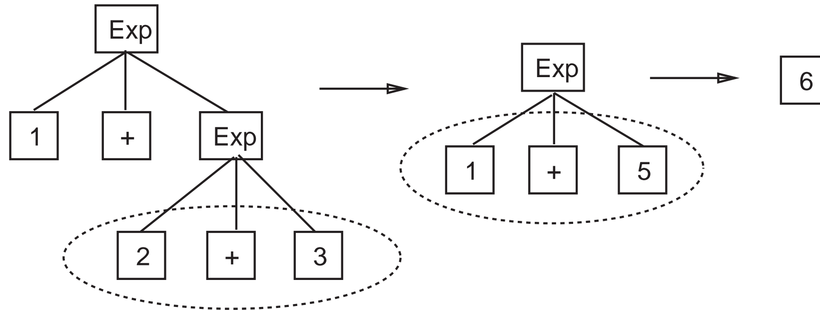

Figura 22: Sequência de interpretação para *1 + 2 + 3*

programa como o visto na Figura 3. Tal programa procederia as corretas reduções, como mostrado na Figura 22.

Na linguagem $\mathcal{SCRIPT}$ nodos possuem construtores implícitos, os quais os identificam de maneira única entre os diversos diferentes nodos que podem aparecer em um programa. Os programas traduzidos disponibilizam para cada nodo construtores explícitos, uma vez que cada possível tipo de nodo é um elemento pertencente ao Tipo Universal, descrito na Seção 5. Durante o processo de tradução, para cada possível tipo de nodo é gerado um identificador único, o qual, posteriormente, será utilizado para identificar o construtor atribuído àquele nodo. Nos programas gerados automaticamente, portanto, nodos, como por exemplo: $[e_1 \ldots e_n]$ são representados como: $\texttt{NODE}x\ e_1 \ldots e_n$, onde x representa um número inteiro que identifica de maneira única este nodo entre os demais nodos gerados no programa traduzido. Observe, na Figura 19, os construtores gerados para denotar os nodos.

9 Descrição Formal da Linguagem *LiLoCa*

Na Seção 8 utilizou-se uma linguagem de programação extremamente simples para descrever o funcionamento do gerador de interpretadores. Esta, entretanto, é uma linguagem de programação por demais desprovida de recursos, de forma que, a fim de demonstrar efetivamente a utilização de $\mathcal{SCRIPT}$, faz-se necessário a proposição de uma linguagem mais poderosa.

Nesta seção é proposta a linguagem de programação *LiLoCa*. Tal linguagem, embora preste-se apenas a fins didáticos, possui a maior parte das construções que caracterizam as grandes linguagem imperativas, como por exemplo, estruturas de repetição, estruturas condicionais, operadores aritméticos e lógicos e permite a declaração de variáveis e de funções.

A fim de prover uma descrição precisa da linguagem *LiLoCa* foram definidos três programas $\mathcal{SCRIPT}$. O primeiro deles trata do aspecto léxico da lingua-

```
GRAMMAR LiLoCa
LEXIS
  UNIT ::= ide | num | true | false | oper | key;

  true  ::= "true": TT;
  false ::= "false": FF;
  num   ::= digit+ : NUMBER;
  digit === "0" .. "9";
  ide   ::= letter+ : QUOTE;
  letter === "a".."z" | "A".."Z" | "_"
  oper  ::= "(" | ")" | "*" | "+" | "-" | "<="
          | ">=" | "<" | ">" | "==" | "!="
          | "{" | "}" | ":=" | ";" | "="
  key   ::= "program" | "var" | "func" | "output" |
            "while" | "do" | "if" | "then" | "else"

END LiLoCa
```

Figura 23: Programa *SCRIPT* que descreve o alfabeto da linguagem *LiLoCa*.

gem, ao passo que o segundo contém a descrição sintática da mesma. O último deles, por sua vez, descreve o significado semântico de cada construção presente na linguagem.

9.1 Definição das Unidades Léxicas da Linguagem

A descrição léxica de uma linguagem é composta pelos símbolos que podem ser encontrados na mesma. Estes símbolos podem ser agrupados em *palavras chaves*, *identificadores* ,*numerais* e *operadores*. Tal descrição permite que sejam gerados analisadores léxicos baseados em autômatos finitos, como os que são produzidos pelo programa *lex* [12]. Estas ferramentas são umas das partes fundamentais de qualquer compilador.

LiLoCa não lida com números de ponto flutuante, possui poucas palavras reservadas e uma quantidade muito limitada de operadores, de forma que sua descrição léxica, vista na Figura 23 é clara e sucinta.

9.2 A Gramática da Linguagem *LiLoCa*

A sintaxe de uma linguagem de programação determina a estrutura dos programas escritos na mesma, isto é, como os diversos elementos léxicos são agrupados. *SCRIPT* permite que *gramáticas livres de contexto* possam ser especificadas para

descrever a estrutura sintática de uma linguagem de programação. Uma gramática livre de contexto consiste em um número finito de *produções*. Cada produção é composta por um símbolo abstrato denominado *não-terminal* em seu lado esquerdo, e por uma seqüência de um ou mais símbolos não-terminais ou *terminais* em seu lado direito. Os símbolos denominados terminais fazem parte do alfabeto léxico da linguagem.

A partir de um símbolo inicial, uma gramática livre de contexto especifica toda uma linguagem, ou seja, o conjunto infinito de possíveis seqüências de terminais que pode resultar da aplicação de sucessivas *regras de produção*. Uma regra de produção consiste em substituir o símbolo não-terminal de uma sentença com a seqüência de símbolos que formam o lado direito de uma produção cujo lado esquerdo é composto por aquele não-terminal. A gramática da linguagem *LiLoCa*, escrita segundo a notação utilizada em $\mathcal{SCRIPT}$, pode ser vista a seguir.

```
GRAMMAR LiLoCa
SYNTAX

  start ::= prog
        ;

  prog  ::= "program" "{" decs ";" coms "}"
        ;

  decs  ::= decs ";" dec
        | dec
        ;

  dec   ::= "var" ide "=" exp
        | "func" ide "(" ide ")" exp
        ;

  coms  ::= coms ";" com
        | com
        ;

  com   ::= ide ":=" exp
        | "output" exp
        | "while" exp "do" "{" coms "}"
        ;

  exp   ::= "if" boolTerm "then" aritExp "else" aritExp
        | exp "==" boolTerm
        | exp "!=" boolTerm
```

```
        | boolTerm
        | false
        | true
        ;

  boolTerm  ::= boolTerm ">" aritExp
            | boolTerm ">=" aritExp
            | boolTerm "<" aritExp
            | boolTerm "<=" aritExp
            | aritExp
            ;

  aritExp   ::= aritExp "+" term
            | aritExp "-" term
            | term
            ;

  term      ::= term "*" factor
            | term "/" factor
            | factor
            ;

  factor    ::= "-" factor
            | ide "(" exp ")"
            | "(" exp ")"
            | num
            | ide
            | "read"

END LiLoCa
```

9.3 Descrição Semântica da Linguagem *LiLoCa*

Enquanto as características léxicas e sintáticas de uma linguagem de programação cuidam de seu aspecto morfológico, a descrição semântica da mesma envolve-se com questões relacionadas ao significado de cada construção presente naquela linguagem. A definição semântica de uma linguagem relaciona-se intimamente com sua definição sintática, uma vez que, em geral, cada construção semanticamente relevante em uma linguagem possui uma regra de produção a partir da qual é produzida, definida na gramática da linguagem, por exemplo, um comando condicional seria gerado pela regra vista na Equação 14.

$$\textbf{command} \rightarrow \text{ if } \textit{expression} \text{ then } \textit{commands} \text{ else } \textit{commands} \tag{14}$$

A descrição semântica de um programa é composta por uma série de funções e por um conjunto de declarações de domínios, conforme fora descrito na Seção 2. Os domínios de uma linguagem de programação definem partes importantes de um sistema computacional, como a entrada e a saída de dados, a memória persistente e os estados alcançados via a execução de comandos. Os domínios definidos para a linguagem *LiLoCa* podem ser vistos nas Equações 15 até 23

$$\begin{aligned}
\textbf{State} &= \textbf{Memory} \times \textbf{Input} \times \textbf{Output} && (15)\\
\textbf{Storeable} &= \textbf{Value} + \textbf{unbound} && (16)\\
\textbf{Memory} &= \textbf{Ide} \rightarrow \textbf{Storeable} && (17)\\
\textbf{Value} &= \textbf{Number} + \textbf{Bool} && (18)\\
\textbf{Input} &= \textbf{Value}* && (19)\\
\textbf{Output} &= \textbf{Value}* && (20)\\
\textbf{A} &= (\textbf{Value} \times \textbf{State}) + \textbf{Error} && (21)\\
\textbf{B} &= \textbf{State} + \textbf{Error} && (22)\\
\textbf{BodyFunc} &= \textbf{State} \rightarrow \textbf{Factor} \rightarrow \textbf{A} && (23)
\end{aligned}$$

Os domínios definidos para a linguagem *LiLoCa* são bastante semelhantes àqueles definidos como exemplo na Seção 2. Neste caso, porém, tem-se a noção de estado decorrente de execução de comandos e de expressões. De acordo com esta definição, a execução de comandos tem como conseqüência única a modificação do estado corrente do sistema computacional, ou a geração de uma mensagem de erro, no caso de comandos mal-formados. O domínio do estado final de um comando é dado pela Equação 22. A execução de uma expressão, por outro lado, pode também modificar o estado do sistema computacional, em decorrência dos chamados "efeitos colaterais", mas caracteriza-se principalmente pela produção de um valor, como está descrito na Equação 21.

Conforme descrito na Seção 8, os interpretadores gerados automaticamente recebem como entrada uma estrutura de dados que descreve a árvore de sintaxe do programa a ser interpretado. O interpretador possui funções para interpretar comandos e diversos tipos de expressões, por exemplo, expressões booleanas e aritméticas. Para cada construção sintática presente na árvore de sintaxe, é invocada uma função específica para aquela construção. A definição de um interpretador em $\mathcal{SCRIPT}$ contém, portanto, a implementação de cada uma destas funções.

```
DEF f factor state : A =
    CASE factor
        /[n] -> (n, state)
        /["read"] ->
            LET (memory, input, output) = state IN
            is-null input -> "Error", (hd input, (memory, tl input, output))
        /[q] ->
            LET (memory, input, output) = state IN
            (memory q EQ ünbound") ->   "Error",
                                        (memory q, (memory, input, output))
        /["-" factor] ->
            LET (value, state1) = f factor state IN
            (NEG value, state1)
        /["(" exp ")"] ->
            LET (value, state1) = e exp state IN
            (value, state1)
        /[q "(" exp ")"] ->
            LET (memory, input, output) = state IN
            (bodyFunc^ (memory q)) state exp
    END
```

Figura 24: Definição semântica de fatores de uma expressão

Nesta seção serão mostradas as funções mais relevantes que descrevem a linguagem *LiLoCa*, sendo inicialmente vistas as construções mais simples, como expressões aritméticas, e a partir delas as construções compostas, como comandos, por exemplo. Uma função semântica em geral usa as definições propostas para construções mais simples. Desta forma, a descrição semântica de uma linguagem de programação assemelha-se a uma hierarquia de funções.

Na Figura 24 vêem-se as definições semânticas que traduzem o significado de fatores de uma expressão. Estas são as unidades mais simples que podem ser encontradas em um programa escrito em *LiLoCa*, pois representam números inteiros, identificadores, aplicação de funções, etc. A função *f* recebe como entrada uma destas construções e o estado do computador. Após a execução, a função retorna, ou uma tupla do tipo *(valor, estado)*, ou uma mensagem de erro, conforme denotado pelo domínio de retorno A. A título de exemplo, a semântica de um identificador pode ser lida da seguinte forma: verifica-se a memória da máquina em busca de uma posição rotulada por aquele identificador. Caso o valor contido naquela posição seja indefinido, retorna-se um erro, do contrário, deve ser retornado o valor armazenado naquele endereço.

Na Figura 25 são mostradas as definições semânticas de termos de expressões.

```
DEF t term state : A =
    CASE term
        /[term1 "*" factor] ->
            LET (value1, state1) = t term1 state IN
            LET (value2, state2) = f factor state1 IN
            (value1 MULT value2, state2)
        /[factor] ->
            LET (value, state1) = f factor state IN
            (value, state1)
    END
```

Figura 25: Definição semântica de termos de expressões

Termos são seqüências de multiplicações aritméticas. A função t recebe um termo e o estado do sistema. De sua execução resulta um novo estado e um valor. A interpretação de uma multiplicação como $t \times f$ se dá em duas etapas: na primeira, cada um dos fatores é avaliado em separado pela respectiva função semântica. Caso de ambas as avaliações resultem números, estes são multiplicados na etapa seguinte do processo. É interessante notar que *LiLoCa* não apresenta o operador de divisão, devido à não implementação do mesmo nesta primeira versão do tradutor $\mathcal{SCRIPT}$.

Na Figura 26 encontram-se as definições semânticas de uma expressão aritmética. Estas são expressões nas quais aparecem operadores aritméticos de adição e subtração. De acordo com a gramática definida para a linguagem *LiLoCa*, tais operadores possuem mais baixa prioridade que o operador de multiplicação.

A semântica de termos de expressões booleanas é mostrada na Figura 27. Tais termos são formados por comparações de ordem, dadas pelos operadores "maior ou igual", "maior", "menor ou igual" e "menor". A semântica de $b > t$, por exemplo, é definida do seguinte modo: encontra-se o valor da expressão b e da expressão t invocando-se as respectivas funções semânticas. Em seguida comparam-se os valores obtidos no passo anterior segundo o operador "maior que".

Na Figura 28 define-se semanticamente uma expressão. Na linguagem *LiLoCa*, expressões são construções que, quando executadas, produzem uma tupla do tipo $(valor \times estado)$ ou um erro de execução. *efeitos colaterais* são produzidos por expressões que alteram o estado do sistema computacional, por exemplo, via a realização de operações de *I/O*. Em *LiLoCa* tem-se expressões condicionais dadas por *if* b *then* x *else* y. No caso deste tipo de expressão, avalia-se inicialmente a expressão booleana b. Caso desta avaliação resulte um valor verdadeiro, procede-se a avaliação semântica da expressão x, do contrário, passa-se à avaliação da expressão y.

```
DEF aExp aritExp state : A =
    CASE aritExp
        /[aritExp1 "+" term] ->
            LET (value1, state1) = aExp aritExp1 state IN
            LET (value2, state2) = t term state1 IN
            (value1 PLUS value2, state2)
        /[aritExp1 "-" term] ->
            LET (value1, state1) = aExp aritExp1 state IN
            LET (value2, state2) = t term state1 IN
            (value1 MINUS value2, state2)
        /[term] ->
            LET (value, state1) = t term state IN
            (value, state1)
    END
```

Figura 26: Definição semântica de expressões aritméticas

```
DEF tBool boolTerm state : A =
    CASE boolTerm
        /[boolTerm1 ">" aritExp] ->
            LET (value1, state1) = tBool boolTerm1 state IN
            LET (value2, state2) = aExp aritExp state1 IN
            (value1 GT value2, state2)
        /[boolTerm1 ">=" aritExp] ->
            LET (value1, state1) = tBool boolTerm1 state IN
            LET (value2, state2) = aExp aritExp state1 IN
            (value1 GE value2, state2)
        /[boolTerm1 "<" aritExp] ->
            LET (value1, state1) = tBool boolTerm1 state IN
            LET (value2, state2) = aExp aritExp state1 IN
            (value1 LT value2, state2)
        /[boolTerm1 "<=" aritExp] ->
            LET (value1, state1) = tBool boolTerm1 state IN
            LET (value2, state2) = aExp aritExp state1 IN
            (value1 LE value2, state2)
        /[aritExp] ->
            LET (value, state1) = aExp aritExp state IN
            (value, state1)
    END
```

Figura 27: Definição semântica de expressões booleanas

```
DEF e exp state : A =
    CASE exp
        /["if" boolTerm "then" aritExp1 "else" aritExp2] ->
            LET (value1, state1) = tBool boolTerm state IN
            value1 EQ TT -> aExp aritExp1 state1, aExp aritExp2 state1
        /[exp1 "==" boolTerm] ->
            LET (value1, state1) = e exp1 state IN
            LET (value2, state2) = tBool boolTerm state1 IN
            (value1 EQ value2, state2)
        /[exp1 "!=" boolTerm] ->
            LET (value1, state1) = e exp1 state IN
            LET (value2, state2) = tBool boolTerm state1 IN
            (value1 NE value2, state2)
        /[boolTerm] ->
            LET (value, state1) = tBool boolTerm state IN
            (value, state1)
        /["false"] -> (FF, state)
        /["true"] -> (TT, state)
    END
```

Figura 28: Definição semântica de expressões

Além de expressões, *LiLoCa* possui comandos. Um comando é uma construção da linguagem de cuja execução resulta uma alteração do estado do sistema computacional. Existem comandos que implementam a atribuição de valor a variável, iterações e operações de escrita de dados no dispositivo de saída. Um comando como *output* exp é interpretado da seguinte forma: em um primeiro passo a expressão *exp* é avaliada. Em seguida o valor resultante desta avaliação é inserido à lista que representa a saída de dados do sistema.

LiLoCa permite a declaração de variáveis e de funções. A semântica de declarações é vista na Figura 30. A definição da semântica de funções é mais complicada, exigindo a implementação de uma segunda função denominada *function*, a qual é mostrada na Figura 31. Uma declaração como *func* f *(*p*) exp* leva à associação do rótulo *f* com a expressão *exp* na memória do sistema.

A função *function* mostrada na Figura 31 representa a entidade que será de fato associada a um identificador de função na memória do sistema. Tal entidade, quando da chamada da função, receberá uma expressão que representa o parâmetro passado e um elemento do domínio *State*, que representa o estado da máquina vigente naquele momento.

Resta, por fim, definir a semântica de um programa escrito em *LiLoCa*. Esta definição pode ser vista na Figura 32. Como pode ser inferido da análise desta

```
DEF c com state : B =
    CASE com
        /[q ":=" exp] ->
            LET (value, (memory, input, output)) = e exp state IN
            (memory q) EQ "unbound" -> "Error",
                                        (memory{q = value}, input, output)
        /["output" exp] ->
            LET (value, (memory, input, output)) = e exp state IN
            (memory, input, value PRE output)
        /["while" exp "do" "{" coms "}"] ->
            LET (value, state1) = e exp state IN
            value IS TT -> (value EQ TT ->
            (LET state2 = cs coms state1 IN
                (c ["while" exp "do" "{" coms "}"] state2)), state1), "Error"
    END
```

Figura 29: Definição semântica de comandos

```
DEF d dec state : B =
    CASE dec
        /["var" q "=" exp] ->
            LET (value, (memory, input, output)) = e exp state IN
            (memory{q = value}, input, output)
        /["func" q1 "(" q2 ")" exp] ->
            LET (memory, input, output) = state IN
            (memory{q1 = (function exp q2 )}, input, output)
    END
```

Figura 30: Definição semântica de declarações

```
DEF function exp q : BodyFunc = LAM state exp1.
                        LET (value, state1) = e exp1 state
                        IN LET (memory, input, output) = state1
                        IN e exp (memory{q = value}, input, output)
```

Figura 31: Definição semântica do corpo de funções

```
DEF elab-prog prog state : B =
    CASE prog
        /["program" "{" decs ";" coms "}"] ->
            cs coms (ds decs state)
    END
```

Figura 32: Definição semântica de um programa escrito em *LiLoCa*

```
DEF ds decs state : B =
    CASE decs
        /[decs1 ";" dec] -> d dec (ds decs1 state)
        /[dec] -> d dec state
    END

DEF cs coms state : B =
    CASE coms
        /[coms1 ";" com] -> c com (cs coms1 state)
        /[com] -> c com state
    END
```

Figura 33: Definição semântica de seqüências de comandos e declarações

função, um programa escrito em *LiLoCa* consiste em uma sequência finita de declarações seguida por uma sequência finita de comandos.

É necessário ainda que sejam definidas funções para interpretar seqüências de comandos e de declarações. Tais funções podem ser vistas na Figura 33.

9.4 Exemplo de programa escrito em *LiLoCa*

Sendo *LiLoCa* uma linguagem de programação, é natural que nela sejam escritos programas de computadores. Um exemplo de tal programa pode ser visto na Figura 34. Este pequeno programa gera uma lista composta dos números inteiros entre 1 e 10 seguidos dos respectivos fatoriais. Este programa, embora simples, contém algumas das construções semânticas mais conhecidas em linguagens imperativas, como operadores de laço (`while`) e estruturas condicionais. Da tradução dos programas $\mathcal{SCRIPT}$ anteriormente descritos resulta um conjunto de programas capazes de realizar a interpretação do programa visto na Figura 34.

```
program
{

    func fat(n) if (n > 0) then (n * fat(n-1)) else 1;

    func inc (n) n+1;

    var sum = 0;

    while (sum < 10) do
    {
        sum := inc (sum);
        output sum;
        output fat(sum)
    }
}
```

Figura 34: Exemplo de programa escrito em *LiLoCa*

10 Conclusão

Deste Projeto Orientado em Computação resultou um Sistema Computacional no qual interpretadores podem ser gerados e, por meio do qual, programas podem ser efetivamente interpretados.

O sistema desenvolvido terá utilidade principalmente acadêmica. Por meio desta ferramenta estudantes de semântica formal poderão exercitar seus conhecimentos teóricos, especificando novas linguagens de programação e utilizando os interpretadores gerados para testar e, eventualmente, corrigir as descrições semânticas por eles formuladas.

Embora tenham sido gerados com sucesso interpretadores para linguagens de programação simples, o sistema desenvolvido neste trabalho possui algumas limitações, as quais podem ser superadas em fases posteriores de seu desenvolvimento. Dentre estas limitações, a principal se refere à inabilidade do sistema em tratar números de ponto flutuante, uma vez que $\mathcal{SCRIPT}$ não possui nenhum domínio relacionado a este tipo de dados.

O tradutor de $\mathcal{SCRIPT}$ para *Haskell* ainda não trata todas as funcionalidades presentes na linguagem $\mathcal{SCRIPT}$. A principal deficiência se refere ao fato deste Sistema não traduzir para *Haskell* as construções de $\mathcal{SCRIPT}$ relacionadas com a programação orientada a objetos. Embora isto não diminua o poder computacional do dispositivo gerado, a inclusão de conceitos de *POO* a uma linguagem funcional é uma interessante linha de pesquisa e a efetiva incorporação desta funcionalidade

ao Sistema seria uma importante contribuição à comunidade científica.

Outro ponto que ainda precisa ser tratado é a tradução em separado de programas *SCRIPT*. Tanto esta linguagem quanto *Haskell* foram desenvolvidas com o objetivo de serem extremamente modulares, entretanto todas as definições semânticas de uma linguagem devem pertencer ao mesmo arquivo fonte para que um interpretador operacional possa ser gerado.

Referências

[1] Fernando M. Q. Pereira. *Implementação de Um Gerador de Interpretadores de Uso Geral.* Relatório Final de Projeto Orientado em Computação I, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, junho, 2001.

[2] Bigonha, Roberto da Silva. *SCRIPT 2.1, An Object Oriented Language for Denotational Semantics.* Relatório Técnico 0xx/00. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, julho, 2000.

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

[4] Taveira, Wendell Figueiredo. *Compilador da Linguagem Funcional Orientada por Objetos SCRIPT para Haskell.* Relatório Final de Projeto de Final de Curso (POCII). Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, fevereiro, 1999.

[5] Gordon, Michael J. C.. *The Denotational Description of Programming Languages* Springer-Verlag, 1979.

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

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

[8] Página web. *Hugs On-line - http://www.haskell.org/hugs/*

[9] Scott, D.S.: *Data Types as Lattices*; SIAM Journal of Computing, Vol. 5, No. 3, (1976)

[10] Welsh, J., Sneeringer, W. J., Hoare, C. A. R.: *Ambiguities and Insecurities in PASCAL*; Software Practice and Experience 7, (1977)

[11] Deitel, H. M., Deitel, P. J., *Java – Como Programar*; Bookman, Porto Alegre, 2001

[12] Johnson, S. *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, N. J., 1978.

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

____
Fernando Magno Quintão Pereira

____
Roberto da Silva Bigonha

____
Mariza A. S. Bigonha