Marco Rodrigo Costa
Orientadora: Mariza Andrade da Silva Bigonha
Co-orientador: Roberto da Silva Bigonha

Compilação de um Cálculo Lambda Estendido para Supercombinadores

Dissertação apresentada ao Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

3 de julho de 2000

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Compilação de um Cálculo Lambda Estendido para Supercombinadores

Marco Rodrigo Costa

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Profa. Mariza Andrade da Silva Bigonha – Orientadora
Departamento de Ciência da Computação – ICEx – UFMG

Prof. Roberto da Silva Bigonha – Co-orientador
Departamento de Ciência da Computação – ICEx – UFMG

Prof. Marcelo de Almeida Maia
Departamento de Ciência da Computação – UFOP

Prof. Carlos Camarão de Figueiredo
Departamento de Ciência da Computação – ICEx – UFMG

Belo Horizonte, 06 de junho de 2000.

Resumo

O presente trabalho trata da compilação de programas $\mathcal{LAMB}$ para $\mathcal{SUPER}$, um Cálculo de Supercombinadores. Ele faz o elo de ligação entre o *Front-End* e o *Back-End* do compilador de $\mathcal{SCRIPT}$. $\mathcal{SCRIPT}$ é uma linguagem funcional com características de linguagens orientadas por objetos. $\mathcal{LAMB}$ representa um Cálculo Lambda Estendido, possui características importantes de linguagens puramente funcionais e pode ser usada como linguagem intermediária para definições em semântica denotacional. Podemos pensar no Cálculo de Supercombinadores também como um Cálculo Lambda estendido. Contudo, as funções não são anônimas, como é o caso das abstrações lambda de $\mathcal{LAMB}$.

Abstract

The present paper attends to compile of programs $\mathcal{LAMB}$ to $\mathcal{SUPER}$, a supercombinations calculus. It makes the link between the Front-End and the Back-End of the compiler of $\mathcal{SCRIPT}$. $\mathcal{SCRIPT}$ is a functional language with caracteristics of object oriented languages. $\mathcal{LAMB}$ represents an Extended Lambda Calculus, has important caracteristics or pure functional languages and is common used as intermediate language for denotational semantics definitions. We can think on supercombinations calculus also as an Extended Lambda Calculus. However, the functions are not anonymous, like $\mathcal{LAMB}$'s lambda abstractions.

Prefácio

Esta dissertação é dedicada aos meus pais, Pedro e Ione, e aos irmãos André e Diana pelo apoio incessante aos meus estudos e pela estabilidade emocional que sempre me proporcionaram.

Agradeço aos professores Roberto e Mariza Bigonha pelo intenso apoio na realização do trabalho.

Agradeço aos amigos, professores e funcionários do DCC.

Agradeço à CAPES pelo apoio financeiro.

Agradeço a todos que me deram alegria durante esses anos.

Sumário

Lista de Figuras

Capítulo 1

Introdução

O presente trabalho trata da compilação de programas $\mathcal{LAMB}$ [Mosses, 1978, Bigonha, 1981] para $\mathcal{SUPER}$, um Cálculo de Supercombinadores [Jones, 1987]. Ele faz o elo de ligação entre o *Front-End* [Oliveira, 1998] e *Back-End* [Maia, 1994] do compilador de $\mathcal{SCRIPT}$ [Bigonha, 1997].

$\mathcal{SCRIPT}$ é uma linguagem cujos programas são formados por módulos que podem importar/exportar funções ou variáveis entre si. Apresenta ainda uma disciplina baseada em equivalência estrutural [Berry and Schwartz, 1979] e poliformismo de inclusão [Cardelli and Wegner, 1985]. Além dessas características, $\mathcal{SCRIPT}$ possui facilidades advindas das linguagens funcionais Miranda [Turner, 1986] e ML [Augustsson, 1984], o que faz com que $\mathcal{SCRIPT}$ se apresente como uma linguagem funcional com recursos para encapsulação, herança e associação dinâmica, podendo ser, portanto, caracterizada como uma linguagem híbrida.

$\mathcal{LAMB}$ é uma linguagem funcional intermediária para definições em semântica denotacional. Ela foi usada como linguagem intermediária na tradução de $\mathcal{SCRIPT}$[Bigonha, 1997]. $\mathcal{LAMB}$ é um Cálculo Lambda [Jones, 1987] estendido, possuindo, portanto, características importantes de linguagens puramente funcionais.

Podemos pensar em $\mathcal{SUPER}$, o Cálculo de Supercombinadores, também como um Cálculo Lambda estendido. Contudo, as funções não são anônimas, como é o caso das abstrações lambda de $\mathcal{LAMB}$. Além disso, os Supercombinadores possuem outra vantagem: se apresentam como uma solução para o problema de variáveis livres[1] existentes em expressões $\mathcal{LAMB}$.

A Figura 1.1 apresenta a arquitetura do compilador para $\mathcal{SCRIPT}$.

[1] Variáveis livres são aquelas variáveis que não estão associadas ou ligadas por nenhum parâmetro de nenhuma abstração lambda que aparece na expressão em questão.

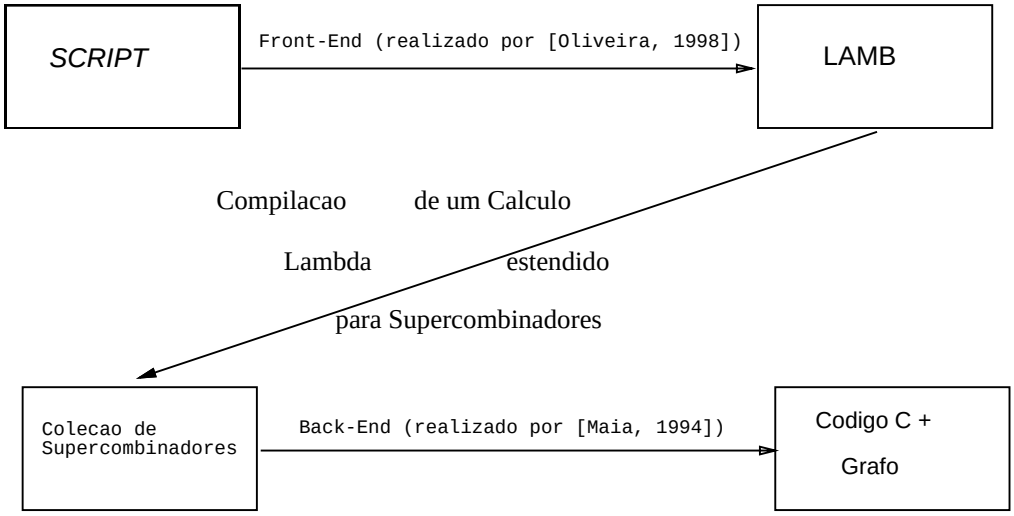

Figura 1.1: Arquitetura do Compilador para $\mathcal{SCRIPT}$

1.1 Perspectiva da Dissertação

Os capítulos deste texto foram organizados de forma simples e progressiva para que o leitor possa realmente acompanhar o desenvolvimento do trabalho. Sendo assim, a disposição ficou a seguinte:

- O Capítulo 2 mostra o contexto geral do compilador para $\mathcal{SCRIPT}$.
- O Capítulo 3 faz uma revisão da literatura sobre Supercombinadores e do processo de transformação para gerá-los: o *lambda-lifting* [Jones, 1987]. Além disso, ainda mostra caminhos alternativos que podem ser seguidos para compilar um Cálculo Lambda.
- O Capítulo 4 mostra a estratégia seguida para a implementação do compilador, inclusive o processo de casamento de padrões, que constitui uma etapa preliminar.
- O Capítulo 5 trata dos pontos mais técnicos do trabalho, mostrando as decisões tomadas durante a implementação do compilador.
- Os exemplos de programas compilados são apresentados no Capítulo 6, com programas gerados para a primeira e segunda etapas.
- Para finalizar, o Capítulo 7 apresenta as contribuições do trabalho realizado e comentários sobre a possibilidade de trabalhos futuros.

Como esta dissertação deve servir de documentação para o trabalho produzido e para algum outro relacionado, algumas informações foram incluídas como apêndices.

- Apêndice A: apresenta a sintaxe de $\mathcal{LAMB}$. Ou seja, programas gerados pelo *Front-End* de $\mathcal{SCRIPT}$ devem atender à forma especificada por essa gramática. Ela representa a linguagem fonte do compilador especificado neste trabalho.

- Apêndice B: como há uma etapa preliminar na geração de código de Supercombinadores, há também uma gramática para representar o conjunto de programas gerados pela primeira fase. Essa "nova" gramática corresponde à linguagem $\mathcal{LAMBASIC}$ e é apresentada nesse apêndice.

- Apêndice C: este apêndice mostra a gramática de Supercombinadores utilizada para a geração final de código deste trabalho. A linguagem foi denominada $\mathcal{SUPER}$.

- Apêndice D: apresenta o formato interno para a representação de programas $\mathcal{LAMB}$ na memória do computador. Algumas idéias para o formato adotado foram retiradas de Peyton Jones e Maia [Jones, 1987, Maia, 1994].

- Apêndice E: o Apêndice E descreve como utilizar o tradutor de $\mathcal{LAMB}$ para $\mathcal{SUPER}$.

- Apêndice G: este apêndice apresenta os supercombinadores auxiliares que podem ser utilizados por supercombinadores gerados durante a tradução.

Capítulo 2

Contexto do Trabalho

Compilar um programa é, na maioria das vezes, uma tarefa complicada. Devido a isso, a técnica de compilação segue a filosofia "DIVIDE AND CONQUER" [Wirth, 1976], onde divide-se o trabalho em várias etapas.

Além da preocupação em traduzir programas, existe também a necessidade de se produzir código de boa qualidade. Para que isto aconteça, é necessário que o projetista da linguagem, além de ter um bom conhecimento das técnicas de compilação, saiba como escolhê-las e utilizá-las para que possa executar seu trabalho satisfatoriamente. Outro requisito também importante diz respeito à forma de apresentação do projeto final. Ele deve ser completo, preciso e sem ambigüidade para que o programador saiba a que sintaxe e semântica obedecer.

Para definir a sintaxe de linguagens, o formalismo denominado *Backus-Naur Form* (BNF) [Aho et al., 1986] tem sido largamente utilizado e se mostrado bastante satisfatório. Contudo, definir a semântica de uma linguagem de programação não é tarefa fácil. Apesar da utilização de linguagens naturais ser predominante para a realização dessa tarefa, elas não servem como **padrão de referência** para a linguagem a ser descrita [Rodrigues, 1993]. Até mesmo as definições formais [Gordon, 1979, Hermano, 1996], que são colocadas como a melhor ferramenta disponível para produzir **padrões de referência** para uma linguagem podem ser precisas, porém difíceis de serem lidas.

Uma proposta para melhoria da legibilidade da descrição de linguagens de programação foi apresentada em [Júnior, 1993]. A solução apresentada combina a naturalidade da prosa informal com o formalismo de métodos formais. Leite propôs uma nova linguagem de descrição, LDS (*Legible Denotational Semantics*) que faz uso do método de programação conhecido como *literate programming* , proposto por Knuth [Knuth, 1984]. Em sua abordagem, Leite [Júnior, 1993] defende um estilo de descrição preciso e consistente e ainda legível e natural. O método se propõe a unir em uma única descrição dois modos de exposição: o modo formal e o modo informal.

No modo informal a apresentação é feita em linguagem natural, baseado na linguagem para formatação de textos LaTeX [Lamport, 1986]. No modo formal a apresentação é feita

com a formulação de equações denotacionais que descrevem a linguagem que se quer especificar. LDS é baseado nas linguagens SSL [Mosses, 1978, Bigonha, 1981] para especificação sintática e SDL [Bigonha, 1981] para especificação semântica.

A linguagem LDS é a parte central de **Xlds**, um ambiente para definição semântica, desenvolvido no Departamento de Ciência da Computação da Universidade Federal de Minas Gerais. Neste sistema, uma definição produz como saída dois subprodutos:

1. um documento formatado em LaTeX com a descrição semântica da linguagem em linguagem natural;

2. um compilador para a linguagem gerado automaticamente a partir da gramática especificada em SSL e da semântica especificada em SDL.

LDS é portanto o resultado da união de uma linguagem para formatação de documentos e duas linguagens de definição formal: SSL e SDL. As decisões do projeto de implementação de LDS e todo o desenvolvimento do compilador é descrito por Walace Rodrigues em [Rodrigues, 1993]. Rodrigues utilizou a linguagem $\mathcal{LAMB}$ [Mosses, 1975] como base semântica para a geração de código.

Com a consolidação de novos paradigmas de linguagens de programação, como o paradigma de programação orientada por objetos (POO) [Meyer, 1988], uma demanda natural por sistemas que atendessem as novas tecnologias se tornou uma obrigação, e como conseqüência, as linguagens SDL e SSL foram unidas e atualizadas dando origem a uma nova linguagem que passou a se chamar $\mathcal{SCRIPT}$ [Bigonha, 1997]. Nesta nova versão da linguagem foram incluídas facilidades tanto do paradigma funcional quanto do paradigma orientado por objetos, caracterizando $\mathcal{SCRIPT}$ como uma linguagem híbrida.

O nosso trabalho está inserido no escopo de um projeto maior que consiste na implementação completa da linguagem $\mathcal{SCRIPT}$ cujas etapas de compilação estão descritas na Seção 2.1.

A Figura 2.1 mostra o contexto geral de LDS e o compilador para $\mathcal{SCRIPT}$.

De acordo com o que foi apresentado, $\mathcal{SCRIPT}$ é, portanto, uma linguagem orientada por objetos cujas descrições de semântica denotacional podem ser efetivamente executadas e depuradas com o auxílio de computadores. $\mathcal{SCRIPT}$ apresenta várias construções oriundas dos sistemas SDL [Bigonha, 1981] e SSL [Mosses, 1978]. Um programa em $\mathcal{SCRIPT}$ é formado por módulos que podem importar e/ou exportar funções ou variáveis entre si; apresenta uma disciplina baseada em equivalência estrutural [Berry and Schwartz, 1979] e polimorfismo de inclusão [Cardelli and Wegner, 1985].

Além dessas características, $\mathcal{SCRIPT}$ possui facilidades advindas das linguagens funcionais Miranda [Turner, 1986] e ML [Augustsson, 1984], o que faz com que $\mathcal{SCRIPT}$ se apresente como uma linguagem funcional com recursos para encapsulação, herança e associação dinâmica. Conseqüentemente, ela provê uma notação adequada para formu-

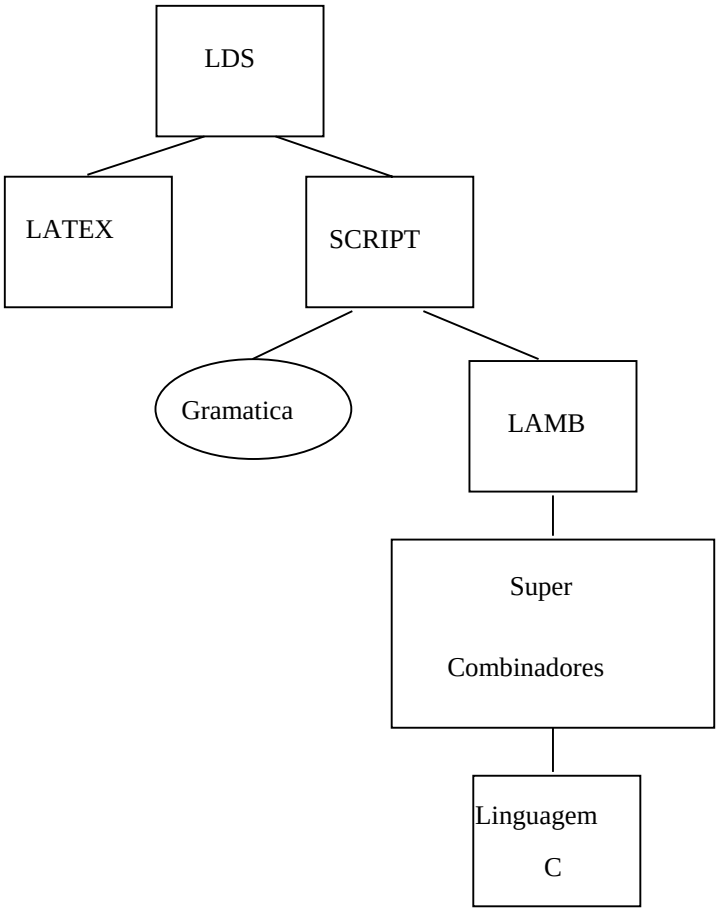

Figura 2.1: Contexto Geral de LDS

lar descrições semânticas denotacionais de linguagens de programação de forma bastante modular.

2.1 O Compilador para $\mathcal{SCRIPT}$

A implementação completa do compilador para $\mathcal{SCRIPT}$ foi dividida em três etapas: a compilação para um cálculo lambda estendido, o processo de *lambda-lifting* [Jones, 1987] e a geração de código. A Figura 1.1 apresenta a sua arquitetura.

A primeira etapa, o *Front-End,* trata da compilação para o cálculo lambda. Ela consiste na compilação da linguagem fonte, $\mathcal{SCRIPT}$, para $\mathcal{LAMB}$. $\mathcal{LAMB}$ é uma linguagem funcional de semântica não-estrita e que provê funções de ordem mais alta [Maia, 1994, Oliveira, 1998]. Ela é utilizada para representar funções matemáticas tais como as funções de definições usadas em semântica denotacional. Esta etapa do trabalho foi desenvolvida por Fabíola Fonseca de Oliveira [Oliveira, 1998].

A segunda etapa desenvolvida trata do gerador de código, o *Back-End.* Ela consiste na

compilação de Supercombinadores[1] para código em linguagem C [Kernighan and Ritchie, 1988] utilizando uma máquina-G estendida [Jones, 1987]. Este trabalho foi executado por Marcelo de Almeida Maia [Maia, 1994].

A terceira etapa, objeto desta dissertação, constitui a etapa intermediária entre o *Front-End* e o *Back-End* do compilador de $\mathcal{SCRIPT}$. Ela consiste na realização do *lambda-lifting,* a tradução das expressões lambda para Supercombinadores [Jones, 1987].

Nas próximas seções são descritas, sucintamente, as duas etapas do compilador já implementadas.

2.1.1 O *Front-End* de $\mathcal{SCRIPT}$

O *Front-End* do compilador recebe como entrada um programa escrito em $\mathcal{SCRIPT}$ e produz como resultado um programa equivalente na linguagem $\mathcal{LAMB}$. Esta fase do compilador compreende: análise léxica, análise sintática, geração da tabela de símbolos, verificação de tipos e a geração do código intermediário, o código $\mathcal{LAMB}$.

Alguns cuidados especiais foram tomados durante a construção do compilador. Houve, por exemplo, a preocupação com o fato de expressões poderem aparecer antes das declarações. Isso acarretou, inclusive, no acréscimo de mais um passo na implementação do compilador, que ficou assim dividido:

Passo 1: processa o texto fonte e realiza as análises léxica e sintática. Faz também a coleta dos domínios[2] e variáveis e suas respectivas inserções na tabela de símbolos. Essa tabela de símbolos encontra-se em uma forma intermediária e é usada como entrada para o segundo passo.

Passo 2: identifica as definições recursivas e as dependências entre as definições. A tabela de símbolos é guardada novamente no final deste passo na mesma forma intermediária utilizada entre o primeiro e o segundo passo.

Passo 3: faz a verificação de tipos e a geração do código final em $\mathcal{LAMB}$ utilizando os resultados dos dois passos anteriores. Neste ponto, o código fonte já foi inspecionado léxica e sintaticamente e os símbolos já foram analisados em termos de suas dependências e recursividades. Portanto a parte semântica pode ser feita.

Cada um desses passos utiliza uma gramática com as produções e rotinas semânticas associadas. Essas gramáticas são submetidas ao YACC[3] que auxilia na produção do código final.

A descrição completa, as decisões de projeto, informações sobre o verificador de tipos e a metodologia usada na implementação do *Front-End* para $\mathcal{SCRIPT}$ estão descritas com detalhes em [Oliveira, 1998].

[1] Supercombinadores são abstrações lambda sem variável livre.
[2] Sinônimo para tipo.
[3] Gerador LALR de analisadores sintáticos.

2.1.2 O Gerador de Código de $\mathcal{SCRIPT}$

O trabalho de Maia [Maia, 1994] foi o ponto de partida para a produção do compilador para $\mathcal{SCRIPT}$. Ele se concentrou no gerador de código. Seu compilador recebe como entrada um programa consistindo de uma coleção de supercombinadores já analisados léxica e sintaticamente, ou seja, um programa correto, e produz como saída um programa equivalente escrito em linguagem C. A estrutura completa do compilador de Supercombinadores para C pode ser vista no Apêndice F de [Maia, 1994].

Para facilitar um melhor entendimento do que seja a implementação de uma linguagem funcional como um todo, é apresentado a seguir um resumo do trabalho de Maia [Maia, 1994]. Ele aborda alguns conceitos fundamentais que foram utilizados durante a implementação do *Back-End* e que podem ser aplicados nas demais fases da compilação de $\mathcal{SCRIPT}$. O primeiro conceito diz respeito à representação interna de programas $\mathcal{LAMB}$. Para descrevê-la, são introduzidas árvores de sintaxe abstrata, grafos de redução abordando representações de células *boxed* e *unboxed* e as estruturas de dados em geral utilizadas em implementações baseadas em redução de grafos. Essas estruturas de dados contêm a expressão, programa funcional, a ser avaliada.

O segundo conceito diz respeito à compilação de casamento de padrões. Nessa etapa, [Maia, 1994] mostra a especificação do compilador de casamento de padrões em $\mathcal{LAMB}$ e apresenta um algoritmo para realizar tal tarefa.

Tendo o problema de casamento de padrões sido resolvido, é necessário avaliar expressões $\mathcal{LAMB}$. Um avaliador executa reduções sucessivas no grafo, envolvendo duas tarefas distintas e bem determinadas:

1. a seleção da próxima expressão reduzível, o *redex* [4];

2. a redução do *redex* selecionado.

Alguns aspectos importantes devem ser considerados na execução dessas reduções: (a) qual deve ser a ordem do processamento das mesmas e (b) qual é o efeito disso no comportamento do programa. O primeiro aspecto levantado refere-se à avaliação *lazy.* Essa avaliação tem dois ingredientes básicos: os argumentos das funções são avaliados somente quando necessários, e; os argumentos devem ser avaliados, no máximo, uma vez. O preço de tal avaliação é o tempo de execução.

A semântica de $\mathcal{LAMB}$ é do tipo *lazy,* o que adiciona expressividade à linguagem principalmente na manipulação de estruturas de dados infinitas. Mesmo assim, existem algumas funções que são estritas em seus argumentos, ou seja, demandam que os mesmos sejam avaliados antes que possam ser executadas. As funções *built-in,* como por exemplo as funções aritméticas *PLUS* e *MINUS* ilustram este caso.

[4] Do inglês *reducible expression.*

Ainda dentro do aspecto da ordem do processamento das reduções, é necessário saber qual é a ordem normal de redução com o objetivo de obter formas normais de expressões e formas normais especiais que possibilitem a avaliação *lazy*. Tais formas são denominadas na literatura de formas normais com cabeça fraca (FNCF) [Jones, 1987].

O processo de redução obedece ao esquema da Figura 2.2.

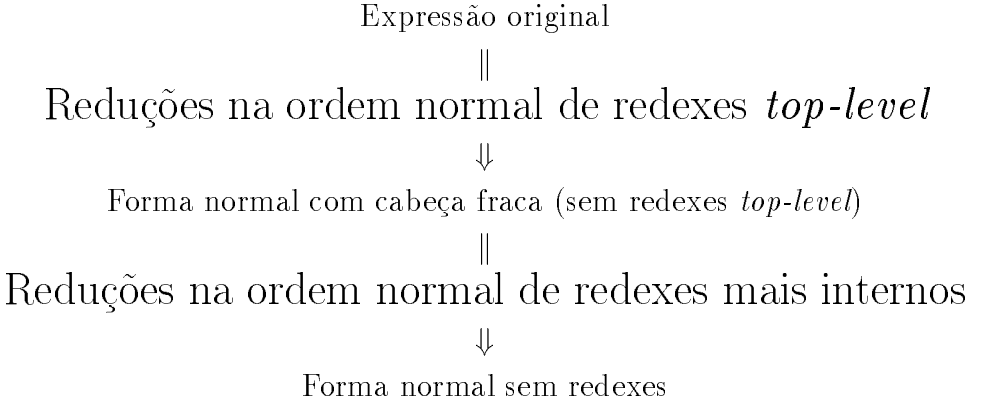

Figura 2.2: Esquema do Processo de Redução

Depois de apresentada e discutida a avaliação *lazy,* é necessário também saber como encontrar o próximo *redex* e como reduzi-lo após tê-lo determinado. Existem várias formas de avaliar uma expressão $\mathcal{LAMB}$. Algumas técnicas são apresentadas em [Maia, 1994] e [Jones, 1987].

Outro conceito importante abordado no trabalho de [Maia, 1994] diz respeito ao algoritmo de *lambda-lifting*[5]. As dificuldades em tratar abstrações lambda e os ganhos com o cálculo de supercombinadores fazem parte de nosso trabalho e serão explicados detalhadamente ao longo deste texto.

A abordagem adotada na implementação do *Back-End* para o compilador $\mathcal{SCRIPT}$, o compilador de Supercombinadores, advém da máquina-G [Johnsson, 1984, Augustsson, 1984, Jones, 1987]. [Maia, 1994] trata, portanto, da tradução de programas $\mathcal{LAMB}$, escritos na forma de um conjunto de supercombinadores, em uma sequência fixa de código, abordagem utilizada para implementar compiladores para outras linguagens funcionais, como por exemplo ML [Augustsson, 1984].

Os elementos que compõem o compilador de Supercombinadores são:

- a linguagem fonte:
  é um programa equivalente ao programa $\mathcal{LAMB}$ original, mas escrito na linguagem de Supercombinadores. Essa linguagem encontra-se formalizada em [Maia, 1994].

[5] Converter abstrações lambda puras em Supercombinadores.

- Os esquemas de compilação:
  são funções que mapeiam as estruturas sintáticas da linguagem de entrada em construções da linguagem objeto. É um estilo denotacional de definir o compilador de Supercombinadores.

- A linguagem objeto:
  conjunto de macros ou simplesmente funções em C.

A compilação de definições de Supercombinadores para o código-G e sua execução pela máquina-G consiste em otimizações do procedimento de instanciação [Jones, 1987]. Para efetuar a compilação da definição de um supercombinador foi definida uma função **F** que recebe a definição do supercombinador como argumento e retorna uma outra definição de função em linguagem C. Essa definição contém:

- o código do corpo do supercombinador compilado em código-G, cujas instruções são definidas em C, caso o mesmo corresponda a uma função ordinária, ou;

- o código do corpo do supercombinador compilado em linguagem C, caso o supercombinador corresponda a uma função especial.

2.1.3 O Trabalho Proposto

Nosso trabalho consistiu especificamente na compilação de programas escritos em $\mathcal{LAMB}$ [Oliveira, 1998] para $\mathcal{SUPER}$, um Cálculo de Supercombinadores. Esse processo é conhecido como *lambda-lifting,* ou seja, a partir de abstrações lambda o objetivo é atingir abstrações equivalentes que não contenham variáveis livres.

A decisão de usar Supercombinadores se deu por motivos de eficiência, uma vez que ao mesmo tempo em que eles resolvem o problema das variáveis livres, eles permitem reduções de multi-argumentos [Maia, 1994, Jones, 1987].

As técnicas utilizadas para a compilação seguiram padrões convencionais para tradução de linguagens funcionais sugeridas por Peyton Jones em [Jones, 1987].

Principais Contribuições

As principais contribuições deste trabalho são o estudo e desenvolvimento de técnicas de compilação para linguagens funcionais e a produção de um componente do compilador da linguagem $\mathcal{SCRIPT}$: o compilador de $\mathcal{LAMB}$ para Supercombinadores.

Capítulo 3

Revisão da Literatura

3.1 Introdução

Este capítulo trata de apresentar e explicar a bibliografia básica estudada [Wakeling, 1998, Turner, 1979, Hindley and Seldin, 1986, Jones, 1987, Jones and Lester, 1994] para que fosse viável o desenvolvimento do presente trabalho.

Primeiramente, foi feito um estudo das abordagens tradicionais de compilação de linguagens funcionais. Em seguida, alguns trabalhos mais específicos também foram estudados para que se pudesse confrontá-los. Como resultado foi possível analisar a proximidade e ligação de cada abordagem estudada com nossa proposta de trabalho.

A compilação de programas utiliza tradicionalmente um código intermediário. Em linguagens funcionais, esse código é, normalmente, a linguagem do Cálculo Lambda mais algumas construções [Jones, 1987]. O Cálculo Lambda é uma coleção de sistemas formais, baseado em uma notação inventada por Alonzo Church nos anos 30 [Hindley and Seldin, 1986]. Ele é designado para capturar as formas em que os operadores ou funções podem ser combinados para formar outros operadores. Na prática, cada sistema lambda apresenta sutis diferenças em sua estrutura gramatical dependendo de sua aplicação. Alguns apresentam símbolos constantes extras ou até mesmo restrições sintáticas embutidas. No compilador para $\mathcal{SCRIPT}$, $\mathcal{LAMB}$ corresponde à linguagem para o Cálculo Lambda estendido.

A partir do momento em que se tem o Cálculo Lambda como código intermediário de uma linguagem funcional, existem caminhos alternativos para se atingir um código final. Por exemplo, uma possibilidade é gerar código para máquina SECD [Landin, 1964, Maia, 1994]. Pode-se ainda adotar uma abordagem semelhante, porém otimizada da máquina SECD, que é a compilação das expressões baseada na idéia de *closures* [Cardelli, 1983, Cardelli, 1984]. Tais abordagens foram muito utilizadas para implementações de linguagens híbridas como o LISP [Sussman, 1982], mas apresentam cuidados especiais tais como o gerenciamento de ambientes para as variáveis de programas.

Outra possibilidade é gerar expressões sem variáveis livres, o que evita o gerenciamento

de ambientes, pois todas as variáveis permanecem em um mesmo ambiente. Para isso, pode-se gerar combinadores [Turner, 1979] ou supercombinadores [Jones, 1987]. O processo de tradução de expressões de um Cálculo Lambda para supercombinadores recebe o nome de *lambda-lifting,* e foi a técnica adotada para este trabalho.

A seguir são apresentados alguns métodos alternativos para compilação de Cálculos Lambda. Iniciamos na Seção 3.2 pela proposta de David Wakeling [Wakeling, 1998] que usa uma Máquina abstratapara implementação do compilador. Em seguida, temos um panorama da proposta original do uso de combinadores [Turner, 1979, Hindley and Seldin, 1986] na Seção 3.3. O ponto mais importante para estudo está concentrado na Seção 3.4, pois trata da abordagem baseada em Supercombinadores adotada por Peyton Jones [Jones, 1987] e escolhida para este trabalho. Para finalizar, é apresentado na Seção 3.5 o resumo do trabalho conjunto [Jones and Lester, 1994] de Peyton Jones e David Lester, cujo estilo de implementação segue o estilo do paradigma funcional.

3.2 Abordagem de David Wakeling

Considerando que linguagens funcionais podem ser bastante inviáveis para a programação em máquinas convencionais devido a sua implementação requerer muita memória para o código e grafo gerados, David Wakeling [Wakeling, 1998] propôs uma nova máquina abstrata para a implementação de linguagens funcionais utilizando computadores convencionais chamada máquina-X. A máquina-X foi projetada de forma que programas possam ser armazenados com a forma compacta de um interpretador, mas que executem com a rapidez de um compilador. Para atingir esse objetivo, é desejável que a máquina abstrata faça bom uso da arquitetura dos computadores modernos, como por exemplo o uso eficiente de registradores, *pipeline* e *cache.* A máquina-X se enquadra perfeitamente nesse esquema.

Na realidade, a máquina-X funciona como um misto da máquina-M e da máquina-G. A máquina-G [Maia, 1994] provê instruções para construção e manipulação de grafos e a máquina-M é uma máquina abstrata de registradores de propósito geral [Wakeling, 1998]. Para se chegar à máquina-X, alguns pontos experimentais foram importantes, como por exemplo a realização de testes com o compilador LML/HBC de Chalmers[Augustsson and Johnsson, 1989]. O compilador de Chalmers primeiramente gera código para a máquina-G e, em seguida, para uma máquina de registradores menos abstrata: a máquina-M. Wakeling ajustou tal compilador a seus trabalhos e chegou a níveis considerados satisfatórios. Nos primeiros resultados de seus trabalhos, os programas requerem apenas 33% do espaço de código de uma implementação de linguagens convencionais, mas executam a 75% da velocidade dos mesmos.

Uma observação importante a ser feita em relação à máquina-X para implementar a linguagem funcional, é que o compilador estático ativado antes do compilador dinâmico (interpretador), divide o código fonte em funções na forma de supercombinadores *(lambda-*

lifted functions) e produz código-X para elas.

Maiores detalhes de como é feita a compilação dinâmica de programas funcionais *lazy* e ainda, da máquina-X, podem ser vistos em [Wakeling, 1998].

3.3 Abordagem Baseada em Combinadores

Muitas abordagens para compilação de linguagens funcionais apresentam um fator indesejado: elas lidam com o gerenciamento de ambientes, ou seja, ao instanciar o corpo de uma abstração lambda existe uma preocupação com o ambiente das variáveis de programa, com seu escopo. E instanciar o corpo de uma abstração lambda, β-redução, é a operação mais essencial na execução de um programa funcional. Por isso ela deve ser implementada da maneira mais eficiente possível.

Devido a este fato, já no final da década de 70 é proposta uma nova abordagem para compilação de linguagens funcionais baseada em combinadores [Turner, 1979, Hindley and Seldin, 1986]. Pode-se dizer que um combinador é uma função pura no sentido que o valor produzido pela aplicação de um combinador a alguns argumentos depende apenas dos valores dos argumentos e não de alguma variável livre. O que se faz, de acordo com essa técnica, é transformar toda abstração lambda em combinador com o objetivo de evitar o gerenciamento de ambientes, já que um combinador só acessa um único ambiente: o de suas variáveis locais. Na verdade, sistemas de combinadores são designados para realizar as mesmas tarefas que sistemas de λ-*calculus,* mas sem usar variáveis livres. O modelo que se propõe para a execução dos combinadores é a redução de grafos.

Apesar de ser uma proposta elegante, várias outras formas de combinadores surgiram como tentativa de otimizar a proposta original de Turner. Dentre elas destacam-se os combinadores SK [Hindley and Seldin, 1986], os multicombinadores categóricos [Lins, 1987] e os supercombinadores [Jones, 1987].

3.4 Abordagem Baseada em Supercombinadores

Os supercombinadores são abstrações lambda especiais que ao mesmo tempo que resolvem o problema das variáveis livres permitem reduções de multi-argumentos. É como uma função em linguagem imperativa com vários parâmetros, que não utiliza variáveis globais e nem provoca efeitos colaterais.

Formalmente, um supercombinador, \$S, de aridade n é uma abstração lambda da forma:

$$LAMx_1.LAMx_2.\cdots LAMx_n.E$$

onde:

1. E não é uma abstração lambda;

2. $S não tem variáveis livres;

3. qualquer abstração lambda em E é um supercombinador;

4. n ≥ 0.

Define-se um *redex* de supercombinador a aplicação de um supercombinador a n argumentos.

São exemplos de supercombinadores: 7, 5 MINUS 2, LAM x.x (identidade), LAM x.x PLUS 1 (incremento) e LAM f.f(LAM x.x PLUS 1).

Não representam supercombinadores: LAM x.z (z ocorre livre), LAM x.x PLUS y (y ocorre livre) e LAM f.f(LAM x.x PLUS f) (LAM x não é supercombinador, pois f ocorre livre nela).

Os supercombinadores sem parâmetros são apenas expressões constantes e são chamados supercombinadores de aridade zero ou, frequentemente, Formas Aplicativas Constantes (FACs) [Jones, 1987]. Portanto: 5, 2 PLUS 3, etc, são exemplos de FACs. Além dessas constantes, qualquer aplicação de função que não receber todos os seus argumentos também é consideradada uma FAC. Um exemplo desse tipo é uma função f que precisa de dois argumentos e recebe apenas um, ou seja: $f : N \rightarrow N \rightarrow N$. $f1$ é uma FAC.

Baseados na definição e nas características de supercombinadores, e ainda considerando as construções da linguagem $\mathcal{LAMB}$, é proposta no Capítulo 4 uma técnica para transformar o Cálculo Lambda estendido em uma coleção de supercombinadores.

Cuidados especiais com abstrações lambda devem ser tomados. Por exemplo, ao serem convertidas para supercombinadores, elas precisam ser analisadas até mesmo quanto à presença de parâmetros redundantes. Definições que apresentem esse comportamento indesejado são tratadas com η-redução[1], e se produzirem definições redundantes, as mesmas devem ser removidas como uma tentativa de otimização do código gerado.

Outra preocupação na geração de supercombinadores é quanto à ordenação dos parâmetros na definição dos supercombinadores. Dependendo da ordem em que eles aparecem, podem surgir simplificações "naturais" no código, as η-reduções. Para contornar esse problema, a primeira atitude é atribuir um nível léxico a cada abstração lambda [Jones, 1987].

Otimizações podem ser feitas em etapas diferentes da compilação de um programa. Na etapa intermediária, antes da geração de supercombinadores, existem alguns pontos susceptíveis a otimização que devem ser tratados. Para se ter uma idéia de um tipo de otimização possível neste nível, imagine trabalhar com abstrações lambda comuns utilizando recursos como a operação *instantiate* [Jones, 1987]. Esta operação é ineficiente pelos seguintes motivos: *instantiate* deve fazer uma análise de caso no *tag* de cada nodo da árvore que representa a função; deve testar, no caso do nodo representar uma variável, se o nodo é o parâmetro formal; um teste similar deve ser feito em cada nodo lambda; como conseqüência, novas instâncias de subexpressões, que não contenham ocorrência livre

[1] η-conversão é o nome dado à operação que remove os parâmetros redundantes de abstrações lambda.

do parâmetro formal, serão construídas enquanto poderiam ser compartilhadas de maneira segura e benéfica [Maia, 1994]. Portanto, pode-se ver que o processo de *lambda-lifting* em si já consiste em uma otimização para a execução de programas funcionais.

Em nosso trabalho, a preocupação maior é com a compilação eficiente de casamento de padrões [Jones, 1987]. Um padrão pode ser: uma constante K, uma variável V ou um padrão construtor [Jones, 1987]. Quando uma função é aplicada a um argumento, esse é avaliado para ver se existe um casamento entre ele e um dos padrões que ele possa representar.

A realização de um casamento de padrões adequado e a eliminação correta de variáveis livres constituem um *lambda lifting* ideal. Qualquer implementação de linguagens de programação deve lidar com o fato de funções e procedimentos poderem possuir variáveis livres. A menos que elas sejam removidas de alguma maneira, um complexo sistema de redução baseado em ambientes deve ser implementado.

3.5 Abordagem de Peyton Jones e David Lester

Peyton Jones e Lester propõem em [Jones and Lester, 1994] a maneira mais popular de se eliminar variáveis livres em linguagens funcionais: o *lambda lifting.* No contexto desse trabalho, eles transformam programas da linguagem Core [Jones and Lester, 1994] em um equivalente onde não há abstrações lambda mais internas, não havendo portanto a presença de variáveis livres.

O que há de mais importante no trabalho de Lester e Jones é a utilização de uma filosofia funcional para implementar o *lambda lifter.* O compilador trabalha em três passos compondo funções: o primeiro passo determina as variáveis livres de cada expressão; o segundo, usa as informações do primeiro passo para realizar uma α-abstração[2] na expressão; o terceiro passo renomeia os supercombinadores e os coleta todos em uma única lista de definições de supercombinadores. Compondo esses três passos tem-se um *lambda lifter* bastante modular, onde a reusabilidade e manutenção de código se fazem marcantemente presentes. A principal desvantagem do trabalho proposto em [Jones and Lester, 1994] é a ausência da otimização chamada "ordenação de parâmetros", não implementada por opção de simplificar o compilador.

3.6 Conclusão

Todas as abordagens apresentadas possuem pontos favoráveis para a utilização neste trabalho, desde que haja algumas adaptações. Mas as abordagens apresentadas nas Seções 3.4 e 3.5 realizam a compilação baseada em Supercombinadores diretamente, ou seja, lidam com nosso mesmo objetivo de tradução, o Cálculo de Supercombinadores. Dentre elas, a

[2]Operação que muda o nome da variável de uma abstração lambda.

que mais se assemelha à nossa proposta é a de Peyton Jones [Jones, 1987], Seção 3.4, por ser um trabalho que explora uma ampla teoria envolvida em processos de tradução dessa natureza.

Capítulo 4

Especificação do Compilador de $\mathcal{LAMB}$

Este capítulo trata da especificação do compilador de $\mathcal{LAMB}$ para supercombinadores. Programas $\mathcal{LAMB}$ são escritos com funções anônimas, abstrações lambda, e com a presença de variáveis livres. São exatamente essas as características a serem eliminadas, portanto pontos essenciais para transformação dos programas. Uma linguagem livre dessas características foi escolhida como linguagem alvo, o cálculo $\mathcal{SUPER}$ de supercombinadores. Os elementos básicos do compilador compreendem: a linguagem fonte, a linguagem objeto e os esquemas de compilação apresentados na Seção 4.1.

O compilador de $\mathcal{LAMB}$ para supercombinadores recebe como entrada programas ou expressões $\mathcal{LAMB}$ geradas pelo *Front-End* [Oliveira, 1998] de $\mathcal{SCRIPT}$ [Bigonha, 1997] e produz como saída uma coleção de supercombinadores gerados, mais supercombinadores auxiliares e mais a expressão a ser avaliada. A expressão a ser avaliada também é um supercombinador. Essa saída servirá como entrada para o gerador de código para Máquina-G [Maia, 1994].

As gramáticas que descrevem as linguagens de programação normalmente não estão na forma ideal para se fazer análise sintática, especialmente quando se trata de submetê-las a geradores de analisadores sintáticos como o YACC [Johnson, 1978]. Algumas transformações são quase sempre necessárias. Tendo em vista este fato, $\mathcal{LAMB}$ [Oliveira, 1998] foi adaptada para a notação BNF padrão e algumas transformações foram efetuadas para retirar as ambigüidades existentes na gramática, como por exemplo, em aplicações de funções. A gramática onde foi baseada a construção de toda especificação do compilador pode ser visualizada no Apêndice A.

O cálculo de supercombinadores para o qual os programas são compilados é representado pela linguagem $\mathcal{SUPER}$. O Apêndice C apresenta a sintaxe BNF de $\mathcal{SUPER}$, que pode também ser encontrada em [Maia, 1994].

4.1 Os Esquemas de Compilação de $\mathcal{LAMB}$

A estratégia utilizada ilustrada na Figura 4.1, consiste, a princípio, na transformação da expressão $\mathcal{LAMB}$ que se quer compilar em um conjunto de definições de supercombinadores mais uma expressão a ser avaliada.

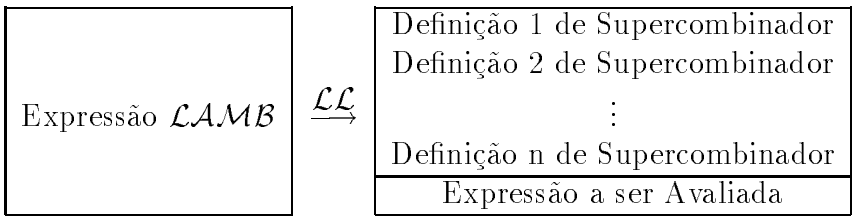

Figura 4.1: Esquema do Processo de Compilação de $\mathcal{LAMB}$

É possível então, pensar no compilador como uma função $\mathcal{LL}$($\mathcal{L}$ambda $\mathcal{L}$ifting), que recebe como argumento uma expressão $\mathcal{LAMB}$ e produz como resultado o código de supercombinadores cuja definição pode ser dada por:

$\mathcal{LL}$(ExpL) =

supercombinadores auxiliares
+ supercombinadores
+ expressão a ser avaliada

ou

$$\mathcal{LL} : ExpL \mapsto ExpS$$

onde: a função $\mathcal{LL}$ é chamada um esquema de compilação [Jones, 1987, Maia, 1994], e utiliza outros esquemas de compilação como funções auxiliares, por exemplo os esquemas representados pela função $\mathcal{P}$ e os esquemas representados pela função $\mathcal{L}$. A notação de esquemas foi adotada por nos permitir expressar as técnicas de compilação de maneira compacta e elegante. $ExpL$ representa uma expressão $\mathcal{LAMB}$. $ExpS$ representa uma expressão $\mathcal{SUPER}$ de supercombinadores. Os supercombinadores auxiliares constituem uma biblioteca de funções que é carregada no final do processo de compilação. Essas funções podem ser utilizadas durante a geração de código. Nesta mesma seção são apresentados todos os supercombinadores auxiliares.

Durante o processo de tradução é necessário que se faça, em um primeiro momento, a compilação de casamento de padrões, representada pelos esquemas $\mathcal{P}$, nas construções onde possa haver a ocorrência interna dos mesmos, para em seguida realizar o *lambda lifting,* representado pelos esquemas $\mathcal{L}$.

A função $\mathcal{P}$ tem tipo:

$$\mathcal{P} : ExpL \mapsto ExpB$$

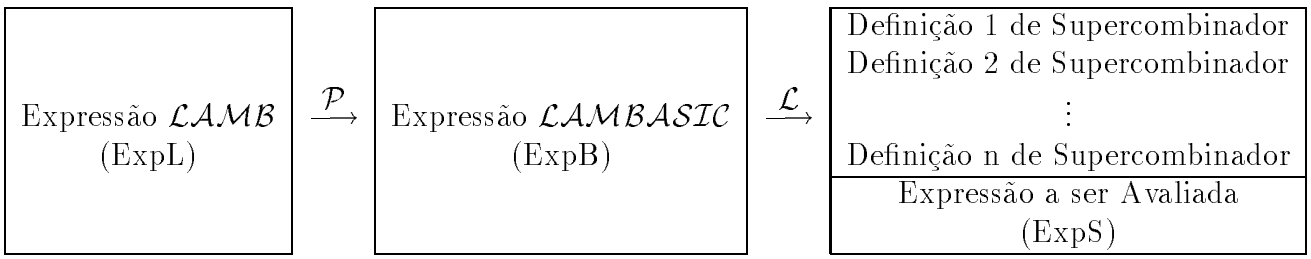

Figura 4.2: Composição dos Esquemas de Compilação $\mathcal{P}$ e $\mathcal{L}$

onde $ExpL$ é o domínio das expressões $\mathcal{LAMB}$ e $ExpB$ o domínio das expressões $\mathcal{LAMBASIC}$, que representa um cálculo lambda estendido mas sem padrões.

O tipo de $\mathcal{L}$ pode ser expresso por:

$$\mathcal{L} : (ExpB \times B \times N) \mapsto (ExpS \times ExpS \times Free).$$

$ExpB$ é o domínio das expressões $\mathcal{LAMBASIC}$ e B é o domínio das variáveis ligadas (ambiente) armazenadas em uma pilha na forma (ide ∈ Ide, n ∈ N, t ∈ Type, x ∈ Ide), onde:

'ide' representa o identificador de tipo 't' ligado no nível léxico 'n' com nome de supercombinador 'x' associado;

N representa o domínio dos inteiros;

Type representa os tipos primitivos Q, N, T e '?'.

Por exemplo, uma tupla (x,3,N,?) indica que o identificador 'x' tem tipo 'N', está ligado no nível léxico 3 (três) e não tem nome de supercombinador associado. Ainda analisando a assinatura de $\mathcal{L}$, o terceiro domínio, identificado por N, representa o nível léxico corrente, fazendo alusão ao escopo da expressão.

$ExpS$ é o domínio das expressões $\mathcal{SUPER}$ de supercombinadores, e $Free$ é um domínio de tuplas (ide ∈ Ide,n ∈ N,t ∈ Type,x ∈ Ide) representando as variáveis livres identificadas na expressão. 'x' é o nome do supercombinador associado a "ide", se houver, senão é o próprio idenficador. Logo, uma tupla (y,2,y,N) indica que o identificador 'y' do tipo 'N' ocorre livre no nível léxico 2 (dois) e é representado por ele mesmo, não é um supercombinador. Já na tupla (y,0,$S1,N), o identificador 'y' ocorre livre no nível léxico 0 (zero), sendo caracterizado como um supercombinador com nome $S1.

Seguindo esse raciocínio, todo o processo de tradução usando um esquema de compilação para realizar o *lambda lifting:* é dado por:

$\mathcal{LL}$(ExpL) =

LET s_0 = definição de supercombinadores auxiliares

LET (s_1,e_1,f_1) = $\mathcal{L}$($\mathcal{P}$(ExpL)) (ϕ) (0)

IN $s_0 \| s_1 \| e_1$

onde ϕ representa, neste caso, uma pilha inicialmente vazia.

De acordo com o esquema acima, para cada expressão $\mathcal{LAMB}$ identificada, deve haver uma chamada à função $\mathcal{LL}$ para que se processe aquela expressão. Este é o processo usual para tradução de programas dessa natureza. Mas na prática, o compilador desenvolvido lida com **módulos** $\mathcal{LAMB}$, e não com uma expressão apenas. Portanto, o processo como um todo ficou mais abrangente. Dentro de um módulo $\mathcal{LAMB}$, pode haver mais que uma expressão sendo compilada.

4.1.1 Supercombinadores Auxiliares

Em $\mathcal{SUPER}$, o único tipo estruturado é a tupla. Portanto, muitas vezes será utilizado o termo tupla ao invés de lista, muito embora o segundo fosse mais apropriado. O conjunto s_0 de supercombinadores auxiliares é composto pelas seguintes funções:

- $EMPTY: esta função testa se uma tupla está vazia.
  $EMPTY: ExpS* $\mapsto$ Booleano
  $EMPTY x* = SIZE x* EQ 0 $->$ TT, FF

- $PREFIX: retorna os 'n' primeiros componentes de uma tupla.
  $PREFIX: Inteiro $\mapsto$ ExpS* $\mapsto$ ExpS*
  $PREFIX n x* = n EQ 0 $->$ ( ) , $HEAD x* PRE $PREFIX (n MINUS 1) $TAIL x*

- $N2Q: converte inteiro para string.
  Exemplo: 123 $\mapsto$ <'1','2','3'>
  $N2Q: Inteiro $\mapsto$ Quotation*
  $N2Q n = (n DIV 10) EQ 0 $->$ $QUOTE n,(($N2Q n DIV 10) AUG $QUOTE (n REM 10))

- $QUOTE: função auxiliar para $N2Q. Converte cada dígito inteiro para seu valor caractere.
  Exemplo: 7 $\mapsto$ '7'
  $QUOTE: Inteiro $\mapsto$ Quotation
  $QUOTE d = d EQ 0 $->$ '0', d EQ 1 $->$ '1',$\cdots$,d EQ 9 $->$ '9','?'

- $T2Q: converte valor booleano para string.
  Exemplo: TT $\mapsto$ <'T','T'>
  $T2Q: Booleano $\mapsto$ Quotation*
  $T2Q t = t EQ TT $->$ < 'T','T' >, t EQ FF $->$ < 'F','F' >, ?

- $Q2Q: converte um string para uma lista de caracteres.
  Exemplo: "teste" $\mapsto$ <'t','e','s','t','e'>
  $Q2Q: Quotation $\mapsto$ Quotation*
  $Q2Q q = (SIZE q EQ 0) $->$ < >, (($HEAD q) CAT ($Q2Q $TAIL q))

- $VAL: força a avaliação estrita de seu argumento.
  $VAL: ExpS $\mapsto$ ExpS
  $VAL x = x

- $HEAD: retorna o primeiro componente de uma tupla, se houver.
  $HEAD: ExpS* $\mapsto$ ExpS
  $HEAD x* = SIZE x* GR 0 $->$ x* EL 1, ?

- $TAIL: retorna os 'n' últimos componentes de uma tupla, se houver.
  $TAIL: ExpS* $\mapsto$ Inteiro $\mapsto$ ExpS*
  $TAIL x* n = SIZE x* LS n $->$ (), (x* EL n) PRE ($TAIL x* n PLUS 1)

- $TRUTH: o booleano correspondente a um string.
  Exemplo: "TT" $\mapsto$ TT
  $TRUTH: Quotation $\mapsto$ Booleano
  $TRUTH q = SIZE q NE 2 $->$?,
  ((q EL 1) EQ 'T') AND ((q EL 2) EQ 'T') $->$ TT,
  ((q EL 1) EQ 'F') AND ((q EL 2) EQ 'F') $->$ FF, ?

4.1.2 Casamento de Padrões

O casamento de padrões consiste em verificar se a estrutura do padrão é a mesma do valor que participará do casamento. Caso o subpadrão seja um identificador, o ambiente deve ser estendido com as ligações entre os subpadrões e o subvalor correspondente. Caso o subpadrão seja uma constante, é necessário testar se ele é igual ao subvalor correspondente. O teste para descobrir qual o padrão em questão segue um algoritmo básico de "unificação" [Aho et al., 1986, Thompson, 1996], mas que não foi utilizado para esta implementação devido às características específicas de $\mathcal{LAMB}$.

A transformação nos padrões de programas $\mathcal{LAMB}$ segue a seguinte ordem: a ocorrência do padrão é substituída por uma nova variável e, em seguida, as ocorrências das variáveis internas ao padrão são substituídas por aplicações de funções seletoras apropriadas na nova variável introduzida para substituir o padrão.

As funções seletoras recebem como parâmetro um valor que deve casar com um padrão e o índice da parte que se quer selecionar desse valor. Tendo em mãos esses dois parâmetros, elas retornam a parte correspondente. Seu protótipo assemelha-se a:

$$SEL(n, tupla)$$

De acordo com a sintaxe de $\mathcal{LAMB}$, um padrão pode ocorrer como parâmetro formal de uma abstração lambda, do lado esquerdo de uma definição de uma construção LET ou DEF, como argumento do lado direito do operador IS ou internamente a outro padrão.

Os padrões mais simples encontrados em $\mathcal{LAMB}$ são:

- o valor *indefinido,* que é representado por "?" e casa com qualquer outro valor;
- uma variável que é representada por um identificador. O operador IS trata um identificador como se fosse "?", casando portanto com qualquer valor;
- outras constantes diferentes do valor especial "?". Essas constantes casam apenas com elas mesmas;
- padrões mais simples combinados com operadores de padrões.

Uma observação importante é que para testar se algum valor é "?", o operador utilizado deve ser o EQ e não o IS. Isso se deve ao fato de "?" casar com qualquer coisa. Como o operador EQ é mais genérico e o operador IS é específico para casamento de padrões, deve-se utilizar o primeiro.

Combinando-se padrões mais simples e operadores de padrões, é possível construir novos padrões na linguagem. A seguir, esses novos padrões são apresentados, onde são considerados $e^{'}, e^{'}_1, e^{'}_2, \cdots, e^{'}_m$ como expressões de padrões, i como identificador e n como uma variável que representa um número inteiro do domínio N.

- LAM ?.? casa com qualquer abstração lambda;
- $(e^{'}_1, \cdots, e^{'}_m)$ casa com tuplas com m componentes;
- $< e^{'}_1, \cdots, e^{'}_m >$ casa com listas com m componentes;
- $[e^{'}_1, \cdots, e^{'}_m]$ casa com nodos da árvore sintática;
- $e^{'}*$ casa com listas com qualquer número de componentes;
- $e^{'}+$ casa com listas com pelo menos um componente;
- $e^{'}_1\ PRE\ e^{'}_2*$ casa com listas com pelo menos um componente;
- $e^{'}_1\ AUG\ e^{'}_2*$ casa com listas com pelo menos um componente;
- $i[n]\ EXT\ e^{'}$ ou $e^{'}_1\ EXT\ e^{'}_2$ casa com tuplas de qualquer tamanho;
- $e^{'}_1\ NODE\ e^{'}_2$ casa com nodos de árvores sintáticas;
- $NUMBER\ e^{'}$ casa com valores numéricos;
- $TRUTH\ e^{'}$ casa com valores booleanos;

- $QUOTE\ e'$ casa com strings *(quotations)*.

De acordo com a semântica de $\mathcal{LAMB}$ é necessário que se faça um casamento entre os componentes do padrão com as subpartes do valor estruturado que participará do casamento.

O algoritmo a seguir deve ser aplicado para cada construção $\mathcal{LAMB}$ onde possa aparecer padrões:

1. identificar quais e quantas são as variáveis internas ao padrão que são efetivamente utilizadas;

2. substituir os padrões por uma variável totalmente nova;

3. no caso de uma subparte do padrão ser uma constante, um código extra deve ser inserido no corpo da construção que está sendo analisada de forma a testar se essa subparte, a constante, é igual ao subvalor correspondente com o qual deverá casar. Caso esse teste retorne TT, o corpo da construção é executado normalmente, caso contrário o valor "?" é retornado;

4. caso o subcomponente do padrão seja uma variável e a mesma seja usada mais de uma vez, seja no lado direito de uma definição LET ou DEF, seja no corpo da construção em questão, devemos:

   - definir um bloco de expressão que substituirá a expressão onde ocorrem os subcomponentes do padrão. Esse bloco de expressão definido será uma expressão LET, e nela são declaradas as variáveis que correspondem aos subcomponentes do padrão. As atualizações dessas variáveis são feitas por meio de chamadas apropriadas das funções seletoras;

   - o corpo desse bloco de expressão deverá ser idêntico àquele da expressão original, a não ser pelo fato das variáveis internas ao padrão serem substituídas pelas variáveis correspondentes declaradas via expressão LET;

5. caso um subcomponente do padrão analisado seja uma variável que é usada apenas uma vez, seja do lado direito de uma definição LET ou DEF ou seja no corpo da construção em questão, devemos substituí-la no corpo da expressão pela aplicação apropriada da função seletora no padrão.

4.2 O Esquema $\mathcal{P}$ de Compilação

O esquema $\mathcal{P}$ representa uma função que recebe uma árvore com padrões e retorna uma outra árvore equivalente sem padrões. O Esquema $\mathcal{P}$, então, é uma função que mapeia expressões em $\mathcal{LAMB}$ para expressões em $\mathcal{LAMBASIC}$. Os padrões podem ocorrer em

três posições distintas em expressões $\mathcal{LAMB}$: (1) como parâmetro formal de abstração lambda, (2) no lado esquerdo de uma definição LET ou DEF e (3) como parâmetro do operador IS.

Para cada uma dessas situações mencionadas, são mostradas, nas Seções 4.2.2, 4.2.3, 4.2.4 e F, as traduções de todos os padrões e expressões que podem aparecer em $\mathcal{LAMB}$ de acordo com sua gramática.

Por questões de clareza e uniformidade, é apresentada, a seguir, a notação adotada na tradução envolvendo casamento de padrões. Essa notação assemelha-se a uma micro-linguagem.

Esquema	::=	LE = LD
LE	::=	$\mathcal{P}[\![\text{exp}]\!]$
LD	::=	Termo^+
Termo	::=	$\mathcal{P}[\![\text{exp}]\!] \mid [\![\text{exp}]\!]$

onde "exp" significa uma seqüência de tokens de $\mathcal{LAMB}$.

É bom observar que:

$$[\![\cdots[\![exp]\!]\cdots]\!] = [\![exp]\!]$$

ou seja, expressões podem aparecer internamente a outras.

4.2.1 Funções Auxiliares

As funções auxiliares não fazem parte do código gerado. Elas são úteis para a formalização dos esquemas de compilação.

novoNome( ): retorna um identificador totalmente novo na expressão.

built_in(operador): retorna o texto correspondente ao operador passado, seja ele unário ou binário.
Exemplo: built_in(PLUS) = PLUS.

$\mathcal{P}_{Pairs}$**:** mapeia uma lista de definições LET, $\mathcal{LAMB}$, em uma lista de pares de expressões $\mathcal{LAMBASIC}$.
$\mathcal{P}_{Pairs}$: defn_listL $\mapsto$ (ExpB,ExpB)*

split(lista): transforma uma lista de pares em uma lista de listas de pares.
split: (ExpB,ExpB)* $\mapsto$ (ExpB*,ExpB*)*
Exemplo:
< ('a','b'),('c','d'),(?,str),('e','f') > $\mapsto$ < < ('a','b'),('c','d') >,< (?,str)> ,< ('e','f') > >

head(lista): retorna a cabeça de uma lista.
head: (ExpB*,ExpB*)* $\mapsto$ (ExpB*,ExpB*)*

tail(lista): retorna a cauda de uma lista.
tail: (ExpB*,ExpB*)* $\mapsto$ (ExpB*,ExpB*)*

cat($pairs_1$,$pairs_2$): concatena duas listas de pares.
cat: (ExpB,ExpB)* $\mapsto$ (ExpB,ExpB)* $\mapsto$ (ExpB,ExpB)*

geraLetList(lista): gera a lista de LETs a partir de uma lista de pares.
geraLetList: (ExpB*,ExpB*)* $\mapsto$ ExpB

geraDefList(lista): gera a lista de DEFs a partir de uma lista de pares.
geraDefList: (ExpB*,ExpB*)* $\mapsto$ ExpB

ggls(lista,string): função chamada por geraLetList. Gera, no parâmetro **string**, a lista efetiva de LETs.

ggds(lista,string): função chamada por geraDefList. Gera, no parâmetro **string**, a lista efetiva de DEFs.

geraLets(lista): recebe uma lista de pares e retorna uma lista de LETs compatível.
geraLets: (ExpB,ExpB)* $\mapsto$ ExpB

geraDefs(lista): recebe uma lista de pares e retorna uma lista de DEFs compatível.
geraDefs: (ExpB,ExpB)* $\mapsto$ ExpB

reorder(listaAgrupada,left,right): gera um par de listas "< left,right >" a partir de uma lista agrupada de pares.
reorder: (ExpB,ExpB)* $\mapsto$ (ExpB*,ExpB*)
Exemplo: < ('a','b'),('c','d'),('e','f') > $\mapsto$ (<'a','c','e'>,<'b','d','f'>)

Algumas dessas funções não foram efetivamente utilizadas na versão final do esquema de compilação. Porém, foram mantidas nesta documentação por se apresentarem como formas alternativas para se obter traduções corretas de programas $\mathcal{LAMB}$.

4.2.2 Padrões como Parâmetro de Abstração Lambda

A idéia básica para a compilação de padrões como parâmetro de abstração lambda pode ser realizada em duas etapas: primeiramente, cria-se uma nova variável para substituir o padrão em questão. Em seguida, toda ocorrência de variáveis internas ao padrão no corpo da abstração lambda são substituídas por chamadas de funções seletoras na nova variável criada.

1. Undefined ('?'):

$\mathcal{P}[\![$`LAM ?.exp`$]\!]$ =
`LET v = novoNome( ) IN` $[\![$`(LAM`$]\!]$ `v` $[\![$`.`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`)`$]\!]$

Exemplo: LAM ?. x PLUS 1 = (LAM v. x PLUS 1)

2. Lista nula ('<>'): como uma lista nula só casa com ela mesma, é necessário então, neste caso, testar se o argumento é uma lista vazia.

$\mathcal{P}[\![$`LAM < >.exp`$]\!]$ =
`LET v = novoNome( ) IN` $[\![$`(LAM`$]\!]$ `v` $[\![$`.$EMPTY`$]\!]$ `v` $[\![-{>}]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`,?)`$]\!]$

onde a função '$EMPTY' é um supercombinador auxiliar.
Exemplo: LAM <>. x PLUS 1 = (LAM v. $EMPTY v − > x PLUS 1 ,?)

3. Identificadores: o identificador é a forma mais simples de um padrão, e na realidade não há o que fazer em relação a casamento de padrão. Mesmo assim, há um avanço na tradução do código.

Há três formas distintas para os identificadores: (1) a forma pura e usual sem discriminador de tipo para o identificador; (2) essa mesma forma, porém com o identificador tipado e rotulado com o operador SPECIAL [Maia, 1994], indicando que a compilação pode ser realizada diretamente para código C, portanto otimizada, e; (3) o identificador simplesmente tipado.

- Identificador ('ide')

$\mathcal{P}[\![$`LAM ide.exp`$]\!]$ = $[\![$`(LAM ide.`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`)`$]\!]$

- Identificador ('SPECIAL ide: type')

$\mathcal{P}[\![$`LAM SPECIAL ide:type.exp`$]\!]$ = $[\![$`(LAM SPECIAL ide:type.`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`)`$]\!]$

- Identificador ('ide: type')

$\mathcal{P}[\![$`LAM ide:type.exp`$]\!]$ = $[\![$`(LAM ide:type.`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`)`$]\!]$

Exemplo:
LAM x:QUOTE. x CAT "sufixo" = (LAM x:QUOTE. x CAT "sufixo")

4. Constante ('atom'): uma constante só casa com ela mesma. Portanto, deve ser adicionado código na tradução para verificar se o argumento passado é a mesma constante. Caso não seja, retorna erro ('?' = "undefined").

$\mathcal{P}[\![$`LAM k.exp`$]\!]$ =
`LET v = novoNome( ) IN` $[\![$`(LAM`$]\!]$ `v` $[\![$`.(k EQ`$]\!]$ `v` $[\![$`) ->`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`,?)`$]\!]$

Exemplo: LAM 3 . x PLUS 1 = (LAM v . (3 EQ v) $->$ x PLUS 1 ,?)

5. Tuplas: os padrões tupla aparecem em $\mathcal{LAMB}$ de três formas e todas elas seguem o algoritmo básico para casamento de padrões apresentado.

5.1- Tupla ('$(pat_1, \cdots, pat_r)$'): esta é a forma mais tradicional de uma tupla e sua tradução é bem simples.

$\mathcal{P}[\![$`LAM` $(p_1, p_2, \cdots, p_r)$`.exp`$]\!]$ = `LET v = novoNome( ) IN`
$[\![$`(LAM`$]\!]$ `v` $[\![$`.`$]\!]$ $\mathcal{P}[\![$`LAM` p_1`.` $\cdots$ `.LAM` p_r`.exp`$]\!]$
$[\![$`(`$]\!]$ `v` $[\![$`EL 1)`$]\!]$ $\cdots$ $[\![$`(`$]\!]$ `v` $[\![$`EL r)`$]\!]$ $[\![$`)`$]\!]$

Exemplo:
LAM (x,y).(x,y) EL 1 = (LAM v.(LAM x.(LAM y.(x,y) EL 1)) (SEL(1,v)) (SEL(2,v)))

5.2- Tupla (ide[n] EXT '$(pat_1, \cdots, pat_r)$'):
neste caso, o primeiro operando de EXT deve ser uma tupla "ide" de tamanho 'n'. O casamento só ocorrerá se a expressão que tenta casar for do tamanho 'n+r'. Logo:

$\mathcal{P}[\![$`LAM ide[n] EXT` $(p_1, \cdots, p_r)$`.exp`$]\!]$ =
`LET v = novoNome( ) IN`
$[\![$`(LAM`$]\!]$ `v` $[\![$`.(SIZE`$]\!]$ `v` $[\![$`) NE (n PLUS` r`) -> ?,`
`( LAM ide.`$]\!]$ $\mathcal{P}[\![$`LAM` p_1`.` $\cdots$ `.` `LAM` p_r`.` `exp`$]\!]$
$[\![$`)($PREFIX n`$]\!]$ `v` $[\![$`)(`$]\!]$ `v` $[\![$`EL n+1)`$]\!]$ $\cdots$ `v` $[\![$`EL n+r)`$]\!]$ $[\![$`)`$]\!]$

onde "$PREFIX" é um supercombinador auxiliar.

5.3- Tupla ('$(pat_1, \cdots, pat_r)$' EXT '$(pat_{r+1}, \cdots, pat_m)$'):
esses padrões tupla só irão casar com expressões de tamanho 'm+r'.

$\mathcal{P}[\![$`LAM` $(p_1, \cdots, p_r)$ `EXT` $(p_{r+1}, \cdots, p_m)$`.exp`$]\!]$ =
`LET v = novoNome( ) IN`
$[\![$`(LAM`$]\!]$ `v` $[\![$`.(SIZE`$]\!]$ `v` $[\![$`NE (`r `PLUS` m`)) -> ?,`
`(`$]\!]$ $\mathcal{P}[\![$`LAM` p_1`.` $\cdots$ `.` `LAM` p_r`.LAM` p_{r+1}`.` `LAM` p_{r+m}`.` `exp`$]\!]$

$[\![)]\!]$ v $[\![$EL 1$)]\!]$ v $[\![\cdots$ v $[\![$EL r$)]\!]$ v $[\![$EL r+1$)]\!]$ $\cdots$ v $[\![$EL r PLUS m$)]\!]$ $[\![)]\!]$

6. Padrões "NUMBER"

6.1- Padrões "NUMBER ?":

$\mathcal{P}[\![$LAM NUMBER ?.exp$]\!]$ =
    LET v = novoNome( ) IN $[\![$(LAM$]\!]$ v $[\![$.$]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![)]\!]$

Exemplo: LAM NUMBER ?.q EL 1 = (LAM v. q EL 1)

6.2- Padrões "NUMBER ide":

$\mathcal{P}[\![$LAM NUMBER ide.exp$]\!]$ =
    LET v = novoNome( ) IN $[\![$(LAM$]\!]$ v $[\![$.(LAM ide.$]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![)($\$$N2Q$]\!]$ v $[\![))]\!]$

onde '$N2Q' é um supercombinador auxiliar.

Exemplo: LAM NUMBER q.q EL 1 = (LAM v.(LAM q. q EL 1) $N2Q v)

7. Padrões "TRUTH":

7.1- Padrões "TRUTH ?":

$\mathcal{P}[\![$LAM TRUTH ?.exp$]\!]$ =
    LET v = novoNome( ) IN $[\![$(LAM$]\!]$ v $[\![$.$]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![)]\!]$

Exemplo:
LAM TRUTH ?. q EL 1 = (LAM v. q EL 1)

7.2- Padrões "TRUTH ide":

$\mathcal{P}[\![$LAM TRUTH ide . exp$]\!]$ =
    LET v = novoNome( ) IN $[\![$(LAM$]\!]$ v $[\![$.(LAM ide.$]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![)($\$$T2Q$]\!]$ v $[\![))]\!]$

onde '$T2Q' é um supercombinador auxiliar.

Exemplo:
LAM TRUTH q . q EL 1 = (LAM v .(LAM q . q EL 1) $T2Q v)

8. Padrões "QUOTE":

8.1- Padrões "QUOTE ?":

$\mathcal{P}[\![$`LAM QUOTE ?.  exp`$]\!]$ =
`LET v = novoNome( ) IN [[(LAM]] v [[.]] P[[exp]] [[)]]`

Exemplo:
LAM QUOTE ? . q CAT "sufixo" = (LAM v.q CAT "sufixo")

8.2- Padrões "QUOTE ide":

$\mathcal{P}[\![$`LAM QUOTE ide .  exp`$]\!]$ =
`LET v = novoNome( ) IN [[(LAM]] v [[.(LAM ide.]] P[[exp]] [[)($Q2Q]] v [[))]]`

onde '$Q2Q' é um supercombinador auxiliar.

Exemplo:
LAM QUOTE q . q CAT "sufixo" = (LAM v.(LAM q. q CAT "sufixo")($Q2Q v))

9. Padrões "VAL":

9.1- Padrões "VAL ?":

$\mathcal{P}[\![$`LAM VAL ?.  exp`$]\!]$ =
`LET v = novoNome( ) IN [[(LAM]] v [[.]] P[[exp]] [[)]]`

Exemplo: LAM VAL ?.x PLUS 1 = (LAM v.x PLUS 1)

9.2- Padrões "VAL ide":

$\mathcal{P}[\![$`LAM VAL ide.exp`$]\!]$ =
`LET v = novoNome( )`
`IN [[(LAM]] v [[.(LAM ide.]] P[[exp]] [[)($VAL]] v [[))]]`

Exemplo: LAM VAL x.x PLUS 1 = (LAM v.(LAM x.x PLUS 1)($VAL v))

10. Padrões "pat_1 PRE pat_2": para este tipo de padrão deve haver um teste para verificar se o argumento é uma lista vazia. Se for, a expressão retornada é *undefined,* senão há o casamento com uma expressão cuja cabeça (*head*) é pat_1 e a cauda (*tail*) é pat_2.

$\mathcal{P}[\![$LAM pat_1 PRE pat_2.exp$]\!]$ =
LET v = novoNome( )
IN $\mathcal{P}$([(LAM] v [.$EMPTY] v [$->$?,
LET pat_1 = $HEAD] v [ALSO LET pat_2 = $TAIL] v [IN exp)])

11. Padrões "pat_1 NODE pat_2": os padrões NODE casam com uma tupla onde o primeiro elemento é uma "quotation" e o segundo é outra tupla. Eles representam o nodo de uma árvore.

$\mathcal{P}[\![$LAM pat_1 NODE pat_2 . exp$]\!]$ =
LET v = novoNome( )
IN $\mathcal{P}$([(LAM] v [.([] v [EL 1) EQ NODE AND ([] v [EL 2) EQ pat_1 $->$
(LAM pat_1.(LAM pat_2.exp)([] v[EL 2)([] v [EL 3),?)])

Exemplo: (LAM "Maria" NODE y . $\cdots$) (NODE,"Maria",(1,2,3)) =
(LAM v.(v EL 1) EQ NODE AND (v EL 2) EQ "Maria" $->$ (LAM "Maria".(LAM y.$\cdots$) (v EL 2)(v EL 3),?)) (NODE,"Maria",(1,2,3)) =
"Maria" EQ "Maria" $->$ (LAM "Maria".(LAM y. $\cdots$)("Maria")(1,2,3)),?

4.2.3 Padrões como Parâmetro de "LET" e de "DEF"

Esta seção mostra como transformar sucessivamente LETs e DEFs gerais em formas mais simples até eventualmente chegar a um cálculo lambda simples. Nesse ponto, é possível realizar o *lambda lifting.*

Se forem permitidos padrões arbitrários no lado esquerdo de definições LETs/DEFs podem surgir complicações. Por exemplo, considere a seguinte expressão:

LET (CONS x xs) = B in exp

onde (CONS x xs) aparece no lado esquerdo da definição.

B pode avaliar para uma lista nula (NIL) ao invés de avaliar para (CONS B_1 B_2), não casando com o padrão pretendido. Nesse caso, algum tipo de erro deve ser relatado. Para garantir que B tenha a mesma forma do padrão especificado, o compilador deve exigir que uma análise de conformidade seja realizada.

Contudo, como a análise de conformidade tem um custo considerável de implementação, ela deve ser evitada sempre que possível. Ela pode ser evitada precisamente em todos os casos onde o casamento de padrões não falhará, como por exemplo, em padrões produtos como as tuplas. Mas como há outros subpadrões que também não falham, surge a necessidade da seguinte definição:

Um padrão 'p' é dito irrefutável se:

1. for uma variável 'v', ou;

2. for um padrão produto da forma $(p_1, \cdots, p_r)$ onde os subpadrões $p_1, \cdots, p_2$ são padrões irrefutáveis.

Em outras palavras, os padrões irrefutáveis são padrões produtos arbitrários com variáveis em seus níveis mais internos. Esses padrões não podem falhar em um sistema com verificador de tipos. No caso de $\mathcal{LAMB}$ essa verificação foi feita no *Front-End* do compilador [Oliveira, 1998].

É sabido que um padrão soma ou até mesmo uma simples constante tornam o padrão refutável, já que existe a possibilidade de não haver casamento. Mas como em $\mathcal{LAMB}$ não existem padrões soma, a análise de conformidade pode ser feita apenas para padrões constantes.

Depois de levantar os problemas que podem surgir devido à presença de padrões refutáveis nas definições LET/DEF, é feita a tradução de expressões propriamente dita.

É natural nesta tradução optar pela geração de cálculo lambda puro por meio de abstrações lambda, já que essa é uma linguagem intermediária natural para implementação de linguagens funcionais. Entretanto, há fortes razões de eficiência para incluir expressões LET/DEF na nossa linguagem alvo ao invés de transformá-las em cálculo lambda ordinário [Jones, 1987]. Especificamente, a transformação de uma expressão LET

$$LET\ ide\ =\ exp_1\ IN\ exp_2$$

na aplicação de uma abstração lambda

$$(LAM\ ide\ .\ exp_2)\ exp1$$

utiliza uma ferramenta poderosa, a abstração lambda, para resolver um caso específico, uma expressão LET. A abstração lambda $(LAM\ ide\ .exp_2)$ poderia ser aplicada a muitos argumentos, mas está sendo efetivamente aplicada a apenas um, exp_1. Ou seja, a generalidade da abstração lambda é desperdiçada em caso de tradução da expressão LET/DEF.

Outro problema é que na maioria das implementações, as expressões LET podem ser avaliadas de forma mais eficiente do que sua equivalente aplicação de abstração lambda. Além de todos esses inconvenientes citados, ainda há um problema relacionado com a transformação de uma expressão DEF em cálculo lambda puro. Para isso, é necessário utilizar o operador de ponto fixo Y [Jones, 1987] para expressar recursão, resultando em uma implementação ineficiente que não é utilizada por nenhum compilador otimizado.

Expostos todos esses problemas, as expressões LET/DEF não serão transformadas para o cálculo lambda na forma de abstrações lambda, mas apenas ajustadas para que seja feito o *lifting* dessas expressões.

Um compilador para casamento de padrões, neste caso, deve proceder da seguinte maneira:

1. forma-se uma lista com novas variáveis criadas para substituir os padrões, ou as mesmas se não existir padrão;

2. cria-se uma tabela com os identificadores que aparecem internamente ao padrão e suas respectivas funções seletoras;

3. gera-se novas definições onde: os lados esquerdos representam a lista de novas variáveis e os lados direitos sendo chamadas às funções seletoras substituindo as variáveis internas aos padrões;

4. substitui-se no corpo do LET/DEF as variáveis internas aos padrões por funções seletoras.

Embora expressões com padrões na presença de LETs e DEFs sejam objeto direto de tradução, elas aparecem em um contexto maior dentro dos programas $\mathcal{LAMB}$. Por causa disso, é apresentado a seguir um esquema de tradução de expressões mais amplas que compreendem tais expressões.

Tradução da expressão "defn_list IN exp"

Nos esquemas para traduzir lista de definições, faz-se necessária a presença de outro esquema auxiliar, o $\mathcal{P}_{Pairs}$, que recebe uma expressão $\mathcal{LAMB}$, lista de LETs ou lista de DEFs, e retorna pares de expressões $\mathcal{LAMBASIC}$. A assinatura desse esquema pode ser encontrada na Seção 4.2.1 e o esquema de tradução aparece logo abaixo.

$\mathcal{P}[\![$`defn_list IN exp`$]\!]$ = $\mathcal{P}[\![$`defn_list`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$

$\mathcal{P}[\![$`defn_list`$]\!]$ = $\mathcal{P}[\![$`defn_list let_list`$]\!]$ | $\mathcal{P}[\![$`defn_list def_list`$]\!]$
| $\mathcal{P}[\![$`let_list`$]\!]$ | $\mathcal{P}[\![$`def_list`$]\!]$

$\mathcal{P}[\![$`defn_list let_list`$]\!]$ = $\mathcal{P}[\![$`defn_list`$]\!]$ $\mathcal{P}[\![$`let_list`$]\!]$

$\mathcal{P}[\![$`defn_list def_list`$]\!]$ = $\mathcal{P}[\![$`defn_list`$]\!]$ $\mathcal{P}[\![$`def_list`$]\!]$

$\mathcal{P}[\![$`let_list`$]\!]$ = $\mathcal{P}[\![$`LET defn_a`$]\!]$

$\mathcal{P}[\![$`def_list`$]\!]$ = $\mathcal{P}[\![$`LET defn_b`$]\!]$

$\mathcal{P}[\![$`LET defn_a`$]\!]$ = $\mathcal{P}[\![$`defn_a`$]\!]$

$\mathcal{P}[\![$`DEF defn_b`$]\!]$ = $\mathcal{P}[\![$`defn_b`$]\!]$

$\mathcal{P}[\![$defn_a$]\!]$ = $\mathcal{P}[\![$defn_a ALSO defn$]\!]$ | $\mathcal{P}[\![$defn$]\!]$

$\mathcal{P}[\![$defn_b$]\!]$ = $\mathcal{P}[\![$defn_b WITH defn$]\!]$ | $\mathcal{P}[\![$defn$]\!]$

$\mathcal{P}[\![$defn_a ALSO defn$]\!]$ =
  LET (left,right) = $\mathcal{P}_{Pairs}[\![$defn$]\!]$
  IN left NE ? $->$ $\mathcal{P}[\![$defn_a$]\!]$ $[\![$LET$]\!]$ left $[\![$=$]\!]$ right $[\![$IN$]\!]$,
    $\mathcal{P}[\![$defn_a$]\!]$ right $[\![->]\!]$ left $[\![$,$]\!]$

$\mathcal{P}[\![$defn_b WITH defn$]\!]$ =
  LET (left,right) = $\mathcal{P}_{Pairs}[\![$defn$]\!]$
  IN left NE ? $->$ $\mathcal{P}[\![$defn_b$]\!]$ $[\![$DEF$]\!]$ left $[\![$=$]\!]$ right $[\![$IN$]\!]$,
    $\mathcal{P}[\![$defn_b$]\!]$ right $[\![->]\!]$ left $[\![$,$]\!]$

$\mathcal{P}[\![$defn$]\!]$ =
  LET (left,right) = $\mathcal{P}_{Pairs}[\![$defn$]\!]$
  IN left NE ? $->$ $[\![$LET$]\!]$ left $[\![$=$]\!]$ right $[\![$IN$]\!]$,
    right $[\![->]\!]$ left $[\![$,$]\!]$

Para traduzir cada tipo de definição, é só seguir os esquemas abaixo:

$\mathcal{P}_{Pairs}[\![$defn$]\!]$ = $\mathcal{P}_{Pairs}[\![$pat = exp$]\!]$

$\mathcal{P}_{Pairs}[\![$pat = exp$]\!]$ =
  $\mathcal{P}_{Pairs}[\![$? = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![<> $ = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$ide = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$SPECIAL ide:type = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$ide:type = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![pat_1$ PRE pat_2 = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![pat_1$ NODE pat_2= exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![(p_1, \cdots, p_r)$ = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$ide EXT $(p_1, \cdots, p_r)$ = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![(p_1, \cdots, p_r)$ EXT $(p_{r+1}, \cdots, p_{r+m})$ = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$k = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$NUMBER ide = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$NUMBER ? = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$TRUTH ide = exp$]\!]$ |
  $\mathcal{P}_{Pairs}[\![$TRUTH ? = exp$]\!]$ |

$\mathcal{P}_{Pairs}[\![$`QUOTE ide = exp`$]\!]$ |
$\mathcal{P}_{Pairs}[\![$`QUOTE ? = exp`$]\!]$ |
$\mathcal{P}_{Pairs}[\![$`VAL ide = exp`$]\!]$ |
$\mathcal{P}_{Pairs}[\![$`VAL ? = exp`$]\!]$

1. "Undefined" ('?'): A tradução de uma expressão LET/DEF com este tipo de padrão fica simples, a saber:

$\mathcal{P}_{Pairs}[\![$`?  = exp`$]\!]$ = $< >$

2. Lista nula ('<>'): $\mathcal{P}_{Pairs}[\![$`< > = exp`$]\!]$ = $<$(?,$[\![$`NOT $EMPTY`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$)$>$

3. Identificadores: Conforme apresentado na Seção 4.2.2, os identificadores também podem aparecer nas mesmas três formas distintas apresentadas para o lado esquerdo de uma definição LET/DEF. da mesma forma, nesses casos, haverá um simples avanço no código sem necessidade de casamento de padrões.

- Identificador ('ide')

  $\mathcal{P}_{Pairs}[\![$`ide = exp`$]\!]$ = $<$($[\![$`ide`$]\!]$,$\mathcal{P}[\![$`exp`$]\!]$)$>$

- Identificador ('SPECIAL ide: type')

  $\mathcal{P}_{Pairs}[\![$`SPECIAL ide:type = exp`$]\!]$ = $<$($[\![$`SPECIAL ide:type`$]\!]$,$\mathcal{P}[\![$`exp`$]\!]$)$>$

- Identificador ('ide : type')

  $\mathcal{P}_{Pairs}[\![$`ide:  type = exp`$]\!]$ = $<$($[\![$`ide:type`$]\!]$,$\mathcal{P}[\![$`exp`$]\!]$)$>$

4. Constante ('atom'): $\mathcal{P}_{Pairs}[\![$`k = exp`$]\!]$ = $<$(?,$[\![$`(k NE`$]\!]$ $\mathcal{P}[\![$`exp`$]\!]$ $[\![$`)`$]\!]$)$>$

5. Tupla

5.1- Tupla ('$(pat_1, \cdots, pat_n)$'):

$\mathcal{P}_{Pairs}[\![(p_1, p_2, \cdots, p_r)$ = exp$]\!]$ =
    LET v = novoNome( )
    IN $\mathcal{P}_{Pairs}$([[LET]] v [[= exp ALSO p_1 = ([[v [[EL 1) ALSO $\cdots$
    ALSO p_r = ([[v [[EL r)]])

onde 'v' é uma variável nova.

5.2- Tupla (ide[n] EXT '$(pat_1, \cdots, pat_n)$'):

$\mathcal{P}_{Pairs}[\![$ide[n] EXT $(p_1, \cdots, p_r)$ = exp$]\!]$ =
    LET v = novoNome( ) IN
    $\mathcal{P}_{Pairs}$([[LET]] v [[= exp
            IN ide = ([[v [[EL1,]] v [[EL 2, $\cdots$,]] v [[EL n)]]
            [[ALSO p_1 = ([[v [[EL n+1) ALSO $\cdots$ ALSO p_r = ([[v [[EL n+r)]])

5.3- Tupla ('$(pat_1, \cdots, pat_r)$ EXT $(pat_{r+1}, \cdots, pat_m)$'):

$\mathcal{P}_{Pairs}[\![(p_1, \cdots, p_r)$ EXT $(p_{r+1}, \cdots, p_{r+m})$ = exp$]\!]$ =
    LET v = novoNome( ) IN
    $\mathcal{P}_{Pairs}$([[LET]] v [[= exp
            ALSO p_1 = ([[v [[EL 1) ALSO $\cdots$ ALSO p_r = ([[v [[EL r)
            ALSO p_{r+1} = ([[v [[EL r+1) ALSO $\cdots$ ALSO p_m = ([[v [[EL m)]])

6. Padrões "NUMBER"

6.1- Padrões "NUMBER ?":

$\mathcal{P}_{Pairs}[\![$NUMBER ? = exp$]\!]$ = $< >$

6.2- Padrões 'NUMBER ide':

$\mathcal{P}_{Pairs}[\![$NUMBER ide = exp$]\!]$ = $<$ ([[ide]],[[$N2Q]] $\mathcal{P}$[[exp]]) $>$

7. Padrões "TRUTH ?":

7.1- Padrões "TRUTH ?":

$\mathcal{P}_{Pairs}[\![$`TRUTH ? = exp`$]\!]$ = $< >$

7.2- Padrões "TRUTH ide":

$\mathcal{P}_{Pairs}[\![$`TRUTH ide = exp`$]\!]$ = $<$([[`ide`]],[[`$T2Q`]] $\mathcal{P}[\![$`exp`$]\!]) >$

8. Padrões "QUOTE"

8.1- Padrões "QUOTE ?":

$\mathcal{P}_{Pairs}[\![$`QUOTE ? = exp`$]\!]$ = $< >$

8.2- Padrões "QUOTE ide":

$\mathcal{P}_{Pairs}[\![$`QUOTE ide = exp`$]\!]$ = $<$([[`ide`]],[[`$Q2Q`]] $\mathcal{P}[\![$`exp`$]\!]) >$

9. Padrões "VAL"

9.1- Padrões "VAL ?":

$\mathcal{P}_{Pairs}[\![$`VAL ? = exp`$]\!]$ = $< >$

9.2- Padrões "VAL ide":

$\mathcal{P}_{Pairs}[\![$`VAL ide = exp`$]\!]$ = $<$([[`ide`]],[[`$VAL`]] $\mathcal{P}[\![$`exp`$]\!]) >$

10. Padrão 'pat_1 PRE pat_2':

$\mathcal{P}_{Pairs}[\![p_1$ `PRE` p_2 `= exp`$]\!]$ =
`LET pairsList =` $\mathcal{P}_{Pairs}($ [[`LET` p_1 `= $HEAD exp ALSO` p_2 `= $TAIL exp`]]
`IN` $<$ `(?,`[[`$EMPTY`]] $\mathcal{P}[\![$`exp`$]\!])$`,pairsList` $>$

11. Padrão 'pat_1 NODE pat_2':

$\mathcal{P}_{Pairs}[\![p_1$ NODE p_2 = exp$]\!]$ =

LET pairsList = $\mathcal{P}_{Pairs}$($[\![$LET p_1 = (exp EL 2) ALSO p_2 = (exp EL 3)$]\!]$)

IN < (?,$\mathcal{P}$($[\![$(exp EL 1) NE NODE OR (exp EL 2) NE p_1 $]\!]$),pairsList >

4.2.4 Padrões como Parâmetro do Operador "IS"

Em uma expressão e_1 IS p_1 é realizado um casamento de padrões a fim de verificar se a expressão e_1 tem a mesma estrutura, forma, descrita pelo padrão p_1 . Caso a expressão e_1 denote um valor que pode ser estruturado de acordo com as regras do padrão p_1, o resultado final da expressão é a constante TT. Caso contrário ou se e_1 for ?, o valor da expressão é a constante booleana FF. Caso e_1 seja $\perp$, o resultado é $\perp$.

1. "Undefined" ('?'): $\mathcal{P}[\![$exp IS ?$]\!]$ = $[\![$TT$]\!]$

Exemplo: 7 PLUS 3 IS ? = TT

2. Lista Nula ('< >'): $\mathcal{P}[\![$exp IS < >$]\!]$ = $[\![$(SIZE$]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![$EQ 0) -> TT, FF$]\!]$

Exemplo: < x > IS < > = (SIZE < x > EQ 0) − > TT , FF

3. Identificadores: Assim como nos casos anteriores, padrões como parâmetros das abstrações lambda e de LETs/DEFs, os identificadores também podem aparecer nas mesmas três formas distintas apresentadas para o lado direito do operador IS. Contudo os identificadores que representam variáveis são tratados pelo operador IS como a constante ?, casando com qualquer valor.

- Identificador ('ide')

  $\mathcal{P}[\![$exp IS ide$]\!]$ = $[\![$TT$]\!]$

- Identificador ('SPECIAL ide : type')

  $\mathcal{P}[\![$exp IS SPECIAL ide : type$]\!]$ = $[\![$TT$]\!]$

- Identificador ('ide : type')

  $\mathcal{P}[\![$exp IS ide : type$]\!]$ = $[\![$TT$]\!]$

Exemplo: (1,2,3) IS x = TT

4. Constante ('atom'): Uma constante literal só casa com ela mesma. Neste caso estamos excluindo o padrão indefinido ? que é um valor especial.

$\mathcal{P}[\![\texttt{exp IS k}]\!] = [\![(]\!]\ \mathcal{P}[\![\texttt{exp}]\!]\ [\![\texttt{EQ k) -> TT, FF}]\!]$

Exemplo: 3 IS 3 = (3 EQ 3) $->$ TT , FF

5. Tuplas:

5.1- Tupla ('$(pat_1, \cdots, pat_r)$'):

$\mathcal{P}[\![\texttt{exp IS } (p_1, p_2, \cdots, p_r)]\!] =$
$\quad [\![\texttt{SIZE}]\!]\ \mathcal{P}[\![\texttt{exp}]\!]\ [\![\texttt{NE r} -> \texttt{FF,}]\!]$
$\quad \mathcal{P}[\![\texttt{(exp EL 1) IS } p_1 \texttt{ AND } \cdots \texttt{ AND (exp EL r) IS } p_r]\!]$

5.2- Tupla (ide[n] EXT '$(pat_1, \cdots, pat_r)$'):

$\mathcal{P}[\![\texttt{exp IS i[n] EXT } (p_1, \cdots, p_r)]\!] =$
$\quad [\![\texttt{SIZE}]\!]\ \mathcal{P}[\![\texttt{exp}]\!]\ [\![\texttt{NE (n PLUS r)} -> \texttt{FF,}]\!]$
$\quad \mathcal{P}[\![\texttt{(exp EL n PLUS 1) IS } p_1 \texttt{ AND } \cdots \texttt{ AND (exp EL n PLUS r) IS } p_r]\!]$

5.3- Tupla ('$(pat_1, \cdots, pat_r)$ EXT $(pat_{r+1}, \cdots, pat_m)$'):

$\mathcal{P}[\![\texttt{exp IS } (p_1, \cdots, p_r) \texttt{ EXT } (p_{r+1}, \cdots, p_m)]\!] =$
$\quad [\![\texttt{SIZE}]\!]\ \mathcal{P}[\![\texttt{exp}]\!]\ [\![\texttt{NE (r PLUS m)} -> \texttt{FF,}]\!]$
$\quad \mathcal{P}[\![\texttt{(exp EL n PLUS 1) IS } p_1 \texttt{ AND } \cdots \texttt{ AND (exp EL r)}$
$\quad \texttt{IS } p_r \texttt{ AND (exp EL r PLUS 1) IS } p_{r+1} \cdots \texttt{ AND (exp EL r PLUS m) IS } p_m\ ]\!]$

6. Padrões "NUMBER"

6.1- Padrões "NUMBER ?":

para este tipo de padrão e "similares", o raciocínio é o mesmo: como os padrões '?' e "ide" casam com qualquer coisa e já houve uma verificação de tipos neste estágio da compilção, o resultado da tradução dessas expressões é a constante "TT".

$\mathcal{P}[\![\texttt{exp IS NUMBER ?}]\!] = [\![\texttt{TT}]\!]$

6.2- Padrões "NUMBER ide":

$\mathcal{P}[\![$exp IS NUMBER ide$]\!]$ = $[\![$TT$]\!]$

7. Padrões "TRUTH"

7.1- Padrões "TRUTH ?":
$\mathcal{P}[\![$exp IS TRUTH ?$]\!]$ = $[\![$TT$]\!]$

7.2- Padrões "TRUTH ide":
$\mathcal{P}[\![$exp IS TRUTH ide$]\!]$ = $[\![$TT$]\!]$

8. Padrões "QUOTE"

8.1- Padrões "QUOTE ?":
$\mathcal{P}[\![$exp IS QUOTE ?$]\!]$ = $[\![$TT$]\!]$

8.2- Padrões "QUOTE ide":
$\mathcal{P}[\![$exp IS QUOTE ide$]\!]$ = $[\![$TT$]\!]$

9. Padrões "VAL"

9.1- Padrões "VAL ?":
$\mathcal{P}[\![$exp IS VAL ?$]\!]$ = $[\![$TT$]\!]$

9.2- Padrões "VAL ide":
$\mathcal{P}[\![$exp IS VAL ide$]\!]$ = $[\![$TT$]\!]$

10. "Operador PRE":

$\mathcal{P}[\![$exp IS pat_1 PRE $pat_2$$]\!]$ =
$\mathcal{P}($ $[\![$$EMPTY exp $->$ FF, $HEAD exp IS pat_1 AND $TAIL exp IS pat_2 $]\!]$)

11. "Operador NODE":

$\mathcal{P}[\![$exp IS pat_1 NODE $pat_2$$]\!]$ =
$[\![$((exp EL 1) EQ NODE) AND ((exp EL 2) EQ pat_1)$]\!]$ $[\![->]\!]$

$\mathcal{P}[\![$(exp EL 3) IS $pat_2]\!]$ $[\![$,FF$]\!]$

4.2.5 Outras Expressões

Módulo $\mathcal{LAMB}$: embora as unidades importantes de compilação sejam as expressões, a compilação é realizada um módulo por vez. Portanto, o esquema $\mathcal{P}$ deve iniciar pela tradução do módulo $\mathcal{LAMB}$, a saber:

$\mathcal{P}[\![$mod_lamb$]\!]$ = $\mathcal{P}[\![$LET ide = exp$]\!]$

$\mathcal{P}[\![$LET ide = exp$]\!]$ = $[\![$LET ide =$]\!]$ $\mathcal{P}[\![$exp$]\!]$

A compilação de uma expressão $\mathcal{LAMB}$ segue o esquema abaixo. A interpretação é bastante direta.

$\mathcal{P}[\![$exp$]\!]$ =
  $\mathcal{P}[\![$LAM pat.exp$]\!]$ |
  $\mathcal{P}[\![$defn_list IN exp$]\!]$ |
  $\mathcal{P}[\![exp_1$ -> exp_2 , exp_3 $]\!]$ |
  $\mathcal{P}[\![exp_1$ op_di exp_2 $]\!]$ |
  $\mathcal{P}[\![$exp IS pat$]\!]$ |
  $\mathcal{P}[\![$op_mono exp$]\!]$ |
  $\mathcal{P}[\![$application$]\!]$ |
  $\mathcal{P}[\![$(tuple_list)$]\!]$ |
  $\mathcal{P}[\![$< tuple_list >$]\!]$ |
  $\mathcal{P}[\![$ide$]\!]$ |
  $\mathcal{P}[\![$atom$]\!]$

1. Expressão Condicional: essa expressão é equivalente a e_2 se e_1 avaliar para TT. É equivalente a e_3 caso e_1 retorne FF. Se e_1 avaliar para ?, o resultado é ?, e se e_1 retornar $\perp$, o resultado de toda expressão é $\perp$.

$\mathcal{P}[\![exp_1$ -> exp_2 , $exp_3]\!]$ = $\mathcal{P}[\![exp_1]\!]$ $[\![$->$]\!]$ $\mathcal{P}[\![exp_2]\!]$ $[\![$,$]\!]$ $\mathcal{P}[\![exp_3]\!]$

2. Operadores Binários:

$\mathcal{P}[\![exp_1$ op_di $exp_2]\!]$ = $\mathcal{P}[\![exp_1]\!]$ *built_in*(op_di) $\mathcal{P}[\![exp_2]\!]$

onde op_di:
"AND", "AUG", "CAT", "DIV", "EL", "EQ", "EXT",
"GE", "GR", "LE", "LS", "MINUS", "MULT",
"NE", "NODE", "OR", "PLUS", "PRE" ou "REM".

A função '*built_in*(op_di)' retorna o texto correspondente ao operador binário.

3. Operadores Unários:

$\mathcal{P}[\![$`op_mono exp`$]\!]$ = *built_in*(`op_mono`) $\mathcal{P}[\![$`exp`$]\!]$

onde op_mono:
"CONC", "HEAD", "NOT", "NUMBER",
"QUOTE", "SIZE", "TAIL" ou "TRUTH".

A função '*built_in*(op_mono)' retorna o texto correspondente ao operador unário ou $HEAD, $TAIL e $TRUTH nos casos dos operadores HEAD, TAIL e TRUTH respectivamente. Esses três últimos operadores não estão presentes em $\mathcal{SUPER}$, e as implementações dos mesmos como supercombinadores pode ser encontrada no Apêndice G.

Exemplos:
1. NOT TT = NOT TT
2. SIZE $< 1, 2 >$= SIZE $< 1, 2 >$

4. Aplicação de Função:

$\mathcal{P}[\![$`application`$]\!]$= $\mathcal{P}[\![$`appl_a`$]\!]$ | $\mathcal{P}[\![$`appl_b`$]\!]$ | $\mathcal{P}[\![$`appl_c`$]\!]$ | $\mathcal{P}[\![$`appl_d`$]\!]$

$\mathcal{P}[\![$`appl_a`$]\!]$= $\mathcal{P}[\![exp_1\ exp_2]\!]$ | $\mathcal{P}[\![$`application exp`$]\!]$

$\mathcal{P}[\![$`appl_b`$]\!]$= $\mathcal{P}[\![exp_1$ `;` $exp_2]\!]$ | $\mathcal{P}[\![$`exp ; application`$]\!]$

$\mathcal{P}[\![$`appl_c`$]\!]$= $\mathcal{P}[\![exp_1$ `CIRC` $exp_2]\!]$ | $\mathcal{P}[\![$`application CIRC exp`$]\!]$

$\mathcal{P}[\![$`appl_d`$]\!]$= $\mathcal{P}[\![exp_1$ `STAR` $exp_2]\!]$ | $\mathcal{P}[\![$`application STAR exp`$]\!]$

5. Aplicação de Função por Justaposição:

$\mathcal{P}[\![exp_1\ exp_2]\!]$ = $\mathcal{P}[\![exp_1]\!]$ $\mathcal{P}[\![exp_2]\!]$

$\mathcal{P}[\![\texttt{application exp}]\!] = \mathcal{P}[\![\texttt{application}]\!]\ \mathcal{P}[\![\texttt{exp}]\!]$

6. Aplicação de Função com Operador ";":

$\mathcal{P}[\![exp_1\ ;\ exp_2]\!] = \mathcal{P}[\![exp_1]\!]\ [\![(]\!]\ \mathcal{P}[\![exp_2]\!]\ [\![)]\!]$

$\mathcal{P}[\![\texttt{exp ; application}]\!] = \mathcal{P}[\![\texttt{exp}]\!]\ [\![(]\!]\ \mathcal{P}[\![\texttt{application}]\!]\ [\![)]\!]$

7. Aplicação de Função com Operador "CIRC":

$\mathcal{P}[\![exp_1$ `CIRC` $exp_2]\!] =$
`LET v = novoNome( )`
`LET w = novoNome( ) IN`
$[\![$`(LAM` $]\!]$ `v` $[\![$`.(LAM` $]\!]$ `w` $[\![$`.(`$]\!]\ \mathcal{P}[\![exp_2]\!]\ [\![$`)($VAL`$]\!]$ `w` $[\![$`))(`$]\!]\ \mathcal{P}[\![exp_1]\!]\ [\![$`)($VAL`$]\!]$ `v` $[\![$`))`$]\!]$

$\mathcal{P}[\![$`application CIRC exp`$]\!] =$
`LET v = novoNome( )`
`LET w = novoNome( ) IN`
$[\![$`(LAM` $]\!]$ `v` $[\![$`.(LAM` $]\!]$ `w` $[\![$`.( `$]\!]\ \mathcal{P}[\![\texttt{exp}]\!]\ [\![$`)($VAL`$]\!]$ `w` $[\![$`))(`$]\!]\ \mathcal{P}[\![\texttt{application}]\!]\ [\![$`)($VAL`$]\!]$ `v` $[\![$`))`$]\!]$

onde 'v' e 'w' são variáveis novas.

Exemplo:
LET f = LAM x.x MULT x ALSO g = LAM x.x PLUS 1
IN (f CIRC g) 3

LET $\cdots$ IN (LAM v.(LAM w.(g($VAL w))) f($VAL v)) 3

8. Aplicação de Função com Operador "STAR":

$\mathcal{P}[\![exp_1$ `STAR` $exp_2]\!] =$
`LET v = novoNome( )`
`LET w = novoNome( )`
`LET t = novoNome( )`
`LET z = novoNome( ) IN`

$[\![$(LAM $]\!]$ v $[\![$.(LAM $]\!]$ z $[\![$.(LAM $]\!]$ w $[\![$.(LAM $]\!]$ t $[\![$.($]\!]$ $\mathcal{P}[\![exp_2]\!]$ $[\![$)($]\!]$ w $[\![$,$]\!]$ t $[\![$))) ($VAL ($]\!]$ z $[\![$EL 1)) ($VAL ($]\!]$ z $[\![$EL 2))) ($]\!]$ $\mathcal{P}[\![exp_1]\!]$ $[\![$)(VAL]\!]$ v $[\![$))$]\!]$

$\mathcal{P}[\![$application STAR exp$]\!]$ =
LET v = novoNome( )
LET w = novoNome( )
LET t = novoNome( )
LET z = novoNome( ) IN
$[\![$(LAM $]\!]$ v $[\![$.(LAM $]\!]$ z $[\![$.(LAM $]\!]$ w $[\![$.(LAM $]\!]$ t $[\![$.($]\!]$ $\mathcal{P}[\![$exp$]\!]$ $[\![$)($]\!]$ w $[\![$,$]\!]$ t $[\![$))) ($VAL ($]\!]$ z $[\![$EL 1)) ($VAL ($]\!]$ z $[\![$EL 2))) ($]\!]$ $\mathcal{P}[\![$application$]\!]$ $[\![$)(VAL]\!]$ v $[\![$))$]\!]$

9. Tuplas:

$\mathcal{P}[\![$(exp_1 , $\cdots$, exp_n)$]\!]$ = $[\![$($]\!]$ $\mathcal{P}[\![exp_1]\!]$ $[\![$,$]\!]$ $\cdots$ $[\![$,$]\!]$ $\mathcal{P}[\![exp_n]\!]$ $[\![$)$]\!]$

10. Listas:

$\mathcal{P}[\![$< exp_1 , $\cdots$, exp_n >$]\!]$ = $[\![$<$]\!]$ $\mathcal{P}[\![exp_1]\!]$ $[\![$,$]\!]$ $\cdots$ $[\![$,$]\!]$ $\mathcal{P}[\![exp_n]\!]$ $[\![$>$]\!]$

11. Identificadores: um identificador "ide" possui o mesmo padrão léxico de formação nas duas linguagens, $\mathcal{LAMB}$ e $\mathcal{LAMBASIC}$.

$\mathcal{P}[\![$ide$]\!]$ = $[\![$ide$]\!]$

Exemplo: Contador = Contador

12. Constantes: se 'k' representa qualquer constante $\mathcal{LAMB}$, então a tradução é simplesmente a identidade, já que o padrão de formação de constantes é o mesmo que em $\mathcal{LAMBASIC}$.

$\mathcal{P}[\![$k$]\!]$ = $[\![$k$]\!]$

Exemplos:
TT = TT
7 = 7
"mensagem" = "mensagem"

4.3 O Esquema $\mathcal{L}$ de Compilação

Finalizada a eliminação de padrões na expressão $\mathcal{LAMB}$ via função $\mathcal{P}$, a função $\mathcal{L}$ é executada para realizar o *lambda lifting* efetivo e completar o processo de compilação. Salientando novamente: o domínio das expressões agora é $\mathcal{LAMBASIC}$, o código alvo é o cálculo de supercombinadores representado por $\mathcal{SUPER}$ e o esquema $\mathcal{L}$ de compilação (veja Seção 4.1) é :

$$\mathcal{L} : ExpB \mapsto B \mapsto N \mapsto ExpS \times ExpS \times Free$$

4.3.1 Funções Auxiliares

Algumas funções auxiliares são utilizadas para que o esquema $\mathcal{L}$ fique coeso e de fácil leitura. As definições de todas essas funções são feitas neste ponto.

insert(b,(ide,n,type,super)): retorna um novo ambiente de variáveis ligadas incluindo '(ide,n,type,super)' no ambiente 'b:B' passado. '(ide,n,type,super)' é o topo e 'b' o restante de uma nova pilha b_1:B;

exclude(f,(ide,n)): retorna o conjunto de variáveis livres no qual exclui-se do conjunto 'f:Free' a quádrupla da forma '(x,k,z,t)' onde x = ide e k = n;

newSuperName( ): gera, a cada chamada, um nome diferente para um supercombinador;

genSuper(nome,f,ide,e): função que gera o string ⟦nome f ide = e⟧, ou seja, a definição textual do supercombinador;

genSuper(nome,e): função que gera o string ⟦nome = e⟧, ou seja, a definição textual do supercombinador associado a um simples identificador;

makeList(f): gera uma lista em ordem crescente de variáveis livres baseada no nível léxico de cada variável. O argumento fornecido deve ser uma lista de variáveis livres. Exemplo: makeList({(a,1,?,⟦ ⟧),(b,1,?,x),(c,3,NUM,⟦ ⟧)}) = ⟦a x c⟧

getSuperName(b,ide): retorna o nome de supercombinador associado a 'ide' em 'b';

getLevel(b,ide): recupera o nível léxico 'n' do identificador 'ide' na pilha 'b', tal que 'n' é o maior nível associado a 'ide';

getType(b,ide): recupera o tipo do identificador 'ide' no ambiente 'b';

genCFun(tipo,nome,p,e): função que gera a definição textual de uma função C-*like* $\mathcal{SUPER}$ obtida a partir da definição de uma função SPECIAL $\mathcal{LAMBASIC}$. O parâmetro "nome" representa o nome gerado para a função C e "tipo" é o tipo

retornado por ela; 'p' é a lista de parâmetros com seus respectivos tipos, e; 'e' é a expressão corpo da função em $\mathcal{SUPER}$. Portanto o resultado é: ⟦CFUN tipo nome p = e⟧.

genTypedParams(free): gera uma lista de parâmetros com seus respectivos tipos obtidos por meio da lista de variáveis livres passada como augumento, "free". Exemplo: genTypedParams({(a,1,?,⟦ ⟧),(b,1,?,x),(c,3,NUM,⟦ ⟧)}) = ⟦void a void x int c⟧.

genTypedParam(type,ide): produz o texto correspondente ao parâmetro sendo traduzido na forma ⟦type ide⟧.

4.3.2 Tradução de um Módulo $\mathcal{LAMBASIC}$

Como a compilação é realizada um módulo por vez, o esquema $\mathcal{L}$ deve iniciar pela tradução de **módulo**, conforme esquema abaixo:

$\mathcal{L}$⟦ mod_lambasic⟧ ϕ 0 = $\mathcal{L}$⟦LET ide = exp⟧ ϕ 0

$\mathcal{L}$⟦LET ide = exp⟧ ϕ 0 =
  LET (s_1, e_1, f_1) = $\mathcal{L}$⟦exp⟧ ϕ 0
  LET s_2 = genSuper("$MOD_LAMB",$e_1$)
  IN (s_1 ‖ s_2 , ⟦ ⟧,f_1)

Expressões $\mathcal{LAMBASIC}$: a compilação de uma expressão $\mathcal{LAMBASIC}$ segue o esquema abaixo. A interpretação é bastante direta.

$\mathcal{L}$⟦exp⟧ =
  $\mathcal{L}$⟦LAM ide_b.exp⟧ |
  $\mathcal{L}$⟦definition IN exp⟧ |
  $\mathcal{L}$⟦exp_1 -> exp_2 , exp_3 ⟧ |
  $\mathcal{L}$⟦exp_1 op_di exp_2 ⟧ |
  $\mathcal{L}$⟦op_mono exp⟧ |
  $\mathcal{L}$⟦application⟧ |
  $\mathcal{L}$⟦(tuple_list)⟧ |
  $\mathcal{L}$⟦< tuple_list >⟧ |
  $\mathcal{L}$⟦ide⟧ |
  $\mathcal{L}$⟦atom⟧

Abstrações Lambda: as abstrações lambda podem ter os identificadores tipados ou atipados. Primeiramente é apresentado um esquema genérico para esses dois tipos de identificadores, e logo a seguir os esquemas para cada um deles.

```
ℒ⟦LAM ide_b.exp⟧ =
  ℒ⟦LAM ide.exp⟧ | ℒ⟦LAM ide:type.exp⟧
```

Abstrações Lambda Atipadas: este tipo de expressão é nossa unidade de compilação, sendo, portanto, a mais importante.

```
ℒ⟦LAM ide.exp⟧ b n =
  LET b1 = insert(b,(ide,n+1,?,?))
  LET (s1, e1, f1) = ℒ⟦exp⟧ (b1) (n+1)
  LET f = exclude(f1,(ide,n+1))
  LET nome = newSuperName( )
  LET livres = makelist(f)
  LET s = s1 || genSuper(nome,livres,ide,e1)
  LET e = nome || livres
  IN (s,e,f)
```

Abstrações Lambda Tipadas: este tipo de expressão $\mathcal{LAMBASIC}$ é usada somente do lado direito de uma função SPECIAL, pois representa um parâmetro da mesma. Portanto, não há geração de supercombinadores nem de função C.

```
ℒ⟦LAM ide:type.exp⟧ b n =
  LET b1 = n NE 0 ->
       insert(b,(ide,n,type,?)),
       insert(b,(ide,n,type,newSuperName()))
  LET (s1, e1, f1) = ℒ⟦exp⟧ (b1) (n)
  IN (s1, e1, f1)
```

Expressões com LET's e/ou DEF's: para estes tipos de expressões há dois casos importantes a se considerar: LETs/DEFs internos a abstrações lambda e LETs/DEFs externos, ou seja, não envolvidos por nenhuma abstração lambda. No primeiro caso os LETs/DEFs são mantidos nos corpos de supercombinadores. No segundo caso, seu nível léxico é igual a 0 (zero), e o LET/DEF correspondente já é um supercombinador.

Antes de compilar uma lista de definições, é realizado um pré-processamento dessa

mesma lista para que se possa estabelecer o ambiente das definições. O tipo de $\mathcal{G}$ é:

$$\mathcal{G} : ExpB \mapsto B \mapsto N \mapsto B$$

ou seja, uma definição $\mathcal{LAMBASIC}$ é avaliada em um ambiente b e um nível léxico n para que se possa estabelecer o ambiente da nova definição.

```
G[[definition]] b n = G[[let_definition]] b n | G[[def_definition]] b n

G[[let_definition]] b n = G[[LET defn]] b n | G[[LET SPECIAL defn]] b n

G[[def_definition]] b n = G[[DEF defn]] b n | G[[DEF SPECIAL defn]] b n

G[[LET defn]] b n = G[[defn]] b n

G[[LET SPECIAL defn]] b n = G[[defn]] b n

G[[DEF defn]] b n = G[[defn]] b n

G[[DEF SPECIAL defn]] b n = G[[defn]] b n

G[[defn]] b n = G[[ide_b = exp]] b n

G[[ide_b = exp]] b n = G[[ide_b]]

G[[ide_b]] b n = G[[ide]] b n | G[[ide :  type]] b n

G[[ide]] b n =
  n NE 0 -> insert(b,(ide,n,?,?)),
  insert(b,(ide,0,?,newSuperName( )))

G[[ide :  type]] b n =
  n NE 0 -> insert(b,(ide,n,type,?)),
  insert(b,(ide,0,type,newSuperName( )))
```

Há ainda um esquema auxiliar para os esquemas $\mathcal{L}$ de definições $\mathcal{LAMBASIC}$, o esquema $\mathcal{I}$. Essa função recebe como parâmetro uma expressão $\mathcal{LAMBASIC}$, mais precisamente um identificador "ide_b", e retorna o identificador propriamente dito. O tipo de $\mathcal{I}$ é, então:

$$\mathcal{I} : ExpB \mapsto ExpB$$

$\mathcal{I}[\![\texttt{ide\_b}]\!]$ = $\mathcal{I}[\![\texttt{ide}]\!]$ | $\mathcal{I}[\![\texttt{ide : type}]\!]$

$\mathcal{I}[\![\texttt{ide}]\!]$ = $[\![\texttt{ide}]\!]$

$\mathcal{I}[\![\texttt{ide : type}]\!]$ = $[\![\texttt{ide}]\!]$

$\mathcal{L}$ para Lista de Definições ("definition IN exp")

$\mathcal{L}[\![\texttt{definition IN exp}]\!]$ b n =
  LET b_1 = $\mathcal{G}[\![\texttt{definition}]\!]$ ϕ n
  LET (s_1,e_1,f_1) = $\mathcal{L}[\![\texttt{definition}]\!]$ b n
  LET b_2 = insert(b,b_1)
  LET (s_2,e_2,f_2) = $\mathcal{L}[\![\texttt{exp}]\!]$ b_2 n
  LET f = exclude($f_1 \cup f_2$,b_1)
  IN ($s_1 \parallel s_2$,$e_1 \parallel e_2$,f)

$\mathcal{L}[\![\texttt{definition}]\!]$ b n = $\mathcal{L}[\![\texttt{let\_definition}]\!]$ b n | $\mathcal{L}[\![\texttt{def\_definition}]\!]$ b n

$\mathcal{L}[\![\texttt{let\_definition}]\!]$ b n = $\mathcal{L}[\![\texttt{LET defn}]\!]$ b n | $\mathcal{L}[\![\texttt{LET SPECIAL defn}]\!]$ b n

$\mathcal{L}[\![\texttt{def\_definition}]\!]$ b n = $\mathcal{L}[\![\texttt{DEF defn}]\!]$ b n | $\mathcal{L}[\![\texttt{DEF SPECIAL defn}]\!]$ b n

$\mathcal{L}[\![\texttt{LET defn}]\!]$ b n =
  LET (s_1,e_1,f_1) = $\mathcal{L}[\![\texttt{defn}]\!]$ b n
  IN n NE 0 -> (s_1,$[\![\texttt{LET}]\!] \parallel e_1 \parallel [\![\texttt{IN}]\!]$,$f_1$),
  (s_1,e_1,f_1)

$\mathcal{L}[\![\texttt{LET SPECIAL defn}]\!]$ b n = $\mathcal{L}[\![\texttt{defn}]\!]$ b n

$\mathcal{L}[\![\texttt{DEF defn}]\!]$ b n =
  LET (s_1,e_1,f_1) = $\mathcal{L}[\![\texttt{defn}]\!]$ b n
  IN n NE 0 -> (s_1,$[\![\texttt{DEF}]\!] \parallel e_1 \parallel [\![\texttt{IN}]\!]$,$f_1$),
  (s_1,e_1,f_1)

```
L[[DEF SPECIAL defn_b]] b n = L[[defn]] b n

L[[defn]]b n = L[[ide_b = exp]] b n


L[[ide_b = exp]] b n =
  LET b1 = G[[ide_b]] b n
  LET (s1,e1,f1) = L[[exp]] b1 n
  LET id = I[[ide_b]]
  LET f2 = exclude(f1,id)
  IN n NE 0 ->
       (s1,id || [[=]] || e1,f2),
       LET nome = getSuperName(b1,id)
       LET t = getType(b1,id)
       IN t EQ ? ->
       LET s2 = genSuper(nome,e1)
       IN (s1 || s2,[[ ]],f2),
       LET livres = genTypedParams(f2)
       LET s2 = genCFun(t,nome,livres,e1)
       LET f3 = exclude(f2,f1)
       IN (s1 || s2,[[ ]],f2)
```

Expressão Condicional: este tipo de expressão $\mathcal{LAMBASIC}$ funciona como em outras linguagens de programação. Caso a expressão e_1 avalie para TT, a expressão e_2 é avaliada. Caso e_1 avalie para FF, e_3 é avaliada. O resultado da expressão é ? quando e_1 produzir ?, e será $\perp$ quando e_1 avaliar para $\perp$.

```
L[[exp1 -> exp2 , exp3]] b n =
     LET (s1, e1, f1) = L[[exp1]] b n
     LET (s2, e2, f2) = L[[exp2]] b n
     LET (s3, e3, f3) = L[[exp3]] b n
     IN (s1 || s2 || s3, [[IF]] || e1 || e2 || e3, f1 ∪ f2 ∪ f3)
```

Exemplo: x GR y $->$ z PLUS w , 10 = ([[ ]],[[IF GR x y PLUS z w 10]],{x,y,z,w})

Operadores Binários: os operadores binários são como funções pré-definidas, e são estritas em seus argumentos, ou seja, exige que seus operandos estejam avaliados. Sua compilação é feita da seguinte maneira:

$\mathcal{L}[\![exp_1$ `op_di` $exp_2]\!]$ `b n =`

`LET` (s_1, e_1, f_1) `=` $\mathcal{L}[\![exp_1]\!]$ `b n`

`LET` (s_2, e_2, f_2) `=` $\mathcal{L}[\![exp_2]\!]$ `b n`

`IN` $(s_1 \parallel s_2, \textit{built-in(op\_di)} \parallel e_1 \parallel e_2, f_1 \cup f_2)$

onde op_di:
"AND", "AUG", "CAT", "DIV", "EL", "EQ", "EXT",
"GE", "GR", "LE", "LS", "MINUS", "MULT",
"NE", "NODE", "OR", "PLUS", "PRE" ou "REM".

Exemplo: 7 PLUS 2 = ([ ],[PLUS [7] [2]],ϕ)

Operadores Unários: como os operadores binários, os unários também exigem a avaliação de seu argumento, e sua tradução é bastante simples.

$\mathcal{L}[\![$`op_mono exp`$]\!]$ `b n =`

`LET` (s_1, e_1, f_1) `=` $\mathcal{L}[\![$`exp`$]\!]$ `b n`

`IN` $(s_1, \textit{built-in(op\_mono)} \parallel e_1, f_1)$

onde op_mono:
"CONC", "HEAD", "NOT", "NUMBER",
"QUOTE", "SIZE", "TAIL" ou "TRUTH".
OBs: os operadores HEAD, TAIL e TRUTH já estão na forma de supercombinadores neste momento.

Exemplos:

1. NOT TT = ([ ],[NOT TT],ϕ)
2. SIZE $< 1,2 >=$ ([ ],[SIZE (1,2)],ϕ)

Aplicação de Função por Justaposição: este foi o único tipo de aplicação de função $\mathcal{LAMB}$ que permaneceu em $\mathcal{LAMBASIC}$ depois da ação da função $\mathcal{P}$.

$\mathcal{L}[\![$`application`$]\!]$ `b n =` $\mathcal{L}[\![exp_1 \ exp_2]\!]$ `b n |` $\mathcal{L}[\![$`application`$]\!]$`b n`

$\mathcal{L}[\![exp_1 \quad exp_2]\!]$ `b n =`

`LET` (s_1, e_1, f_1) `=` $\mathcal{L}[\![exp_1]\!]$ `b n`

`LET` (s_2, e_2, f_2) `=` $\mathcal{L}[\![exp_2]\!]$ `b n`

`IN` $(s_1 \| s_2, [\![AP]\!] \ \| e_1 \| e_2, f_1 \cup f_2)$

$\mathcal{L}[\![$application exp$]\!]$ b n =
LET (s_1, e_1, f_1) = $\mathcal{L}[\![$application$]\!]$ b n
LET (s_2, e_2, f_2) = $\mathcal{L}[\![exp_2]\!]$ b n
IN $(s_1 \| s_2, [\![AP]\!] \ \| e_1 \| e_2, f_1 \cup f_2)$

Componentes de Expressões Estruturadas: Os esquemas seguintes são utilizados como esquemas auxiliares para a tradução de listas e tuplas. Para os dois tipos de expressões a tradução é idêntica.

$\mathcal{L}[\![$tuple_list$]\!]$ b n = $\mathcal{L}[\![$exp_list$]\!]$ b n | $\mathcal{L}[\![\ ]\!]$ b n

$\mathcal{L}[\![$exp_list$]\!]$ b n = $\mathcal{L}[\![$exp_list , exp$]\!]$ b n | $\mathcal{L}[\![$exp$]\!]$ b n

$\mathcal{L}[\![\ ]\!]$ b n = ($[\![\ ]\!]$, $[\![\ ]\!]$,{ })

$\mathcal{L}[\![$exp_list , exp$]\!]$ b n =
LET (s_1, e_1, f_1) = $\mathcal{L}[\![$exp_list$]\!]$ b n
LET (s_2, e_2, f_2) = $\mathcal{L}[\![$exp$]\!]$ b n
IN $(s_1 \ \| \ s_2, e_1 \ \| \ [\![,]\!] \ \| \ e_2, f_1 \cup f_2)$

Tuplas: cada expressão componente de uma tupla pode gerar um supercombinador e o conjunto de variáveis livres é a união das variáveis livres produzidas pela tradução de cada expressão da tupla.

$\mathcal{L}[\![$(tuple_list)$]\!]$ b n =
LET (s_1, e_1, f_1) = $\mathcal{L}[\![$tuple_list$]\!]$ b n
IN $(s_1, [\![(]\!] \ \| \ e_1 \ \| \ [\![)]\!], f_1)$

Listas As listas são tratadas, em $\mathcal{SUPER}$, como tuplas. Portanto, sua compilação é idêntica à tradução das tuplas.

$\mathcal{L}[\![$< tuple_list >$]\!]$ b n =
LET (s_1, e_1, f_1) = $\mathcal{L}[\![$tuple_list$]\!]$ b n
IN $(s_1, [\![(]\!] \ \| \ e_1 \ \| \ [\![)]\!], f_1)$

Identificadores: todo identificador ocorre livre nele mesmo. Portanto, o esquema $\mathcal{L}$ o retorna como parte da tradução e para a lista de variáveis livres f.

```
ℒ⟦ide⟧ b n =
    LET n1 = getLevel(b,ide)
    LET t = getType(b,ide)
    IN n EQ 0 ->
    LET nome = getSuperName(b,ide) IN ([ ],nome,{(ide,n1,t,nome)}),
    ([ ],⟦ide⟧,{(ide,n1,t,⟦ide⟧)})
```

onde "ide" possui o mesmo padrão léxico de formação nas duas linguagens.

Exemplo: Contador = ([ ],⟦Contador⟧,{(Contador,0,'?',$S1)})

Constantes: constantes são denominadas FACs (Formas Aplicativas Constantes) [Jones, 1987], ou seja, são supercombinadores de aridade 0 (zero). O nível léxico de uma constante é igual a 0 (zero), e não há geração de supercombinadores.

```
ℒ⟦k⟧ b n = ([ ],⟦k⟧,φ)
```

onde 'k' representa qualquer constante $\mathcal{LAMB}$.

Exemplos:
TT = ([ ],⟦TT⟧,ϕ)
7 = ([ ],⟦7⟧,ϕ)
"mensagem" = ([ ],⟦"mensagem"⟧,ϕ)

4.4 Conclusão

Características como a presença de padrões na linguagem $\mathcal{LAMB}$, fazem com que o processo de *lambda lifting* seja realizado após um passo importante de tradução: o casamento de padrões. Colocado isso, é possível identificar o processo de tradução de $\mathcal{LAMB}$ para supercombinadores como uma composição de tarefas, mais especificamente como uma composição de funções: a função $\mathcal{L}$, que realiza o *lambda lifting* efetivo, aplicada ao resultado da função $\mathcal{P}$, que realiza o casamento de padrões.

Capítulo 5

Implementação do Compilador de $\mathcal{LAMB}$

Este capítulo trata da implementação do tradutor de programas $\mathcal{LAMB}$ para programas formados por um conjunto de supercombinadores da linguagem $\mathcal{SUPER}$. Como dito no Capítulo 4, os programas $\mathcal{LAMB}$ podem conter funções anônimas: as abstrações lambda. Além disso, ainda podem apresentar expressões com variáveis livres. Para solucionar esses inconvenientes, é necessário que se faça o processo de *lambda-lifting*. Contudo, esse processo de tradução não pode ser realizado diretamente para a obtenção de programas $\mathcal{SUPER}$. Isso se deve ao fato de programas $\mathcal{LAMB}$ poderem apresentar padrões em alguns tipos de expressões. Eles aparecem, mais especificamente: como argumentos de operadores como o IS, que faz análise de conformidade, o LAM, indicativo de uma abstração lambda, e LETs/DEFs que realizam amarração[1]. A presença de tais padrões, então, inviabilizam o processo de *lambda-lifting* escolhido para este trabalho porque os únicos padrões aceitáveis para que se tenha um processo ideal de tradução são os identificadores. Sendo assim, o compilador foi desenvolvido em duas etapas. Na primeira, as constantes e padrões estruturados foram eliminados, ou melhor dizendo, transformados. Na segunda etapa foi feito o *lambda-lifting* efetivo. As Seções 5.1 e 5.2 descrevem, respectivamente, em detalhes cada uma dessas etapas de implementação.

5.1 Estruturas de Dados utilizadas na Tradução de Programas $\mathcal{LAMB}$ para $\mathcal{LAMBASIC}$

A primeira etapa da tradução se encarrega apenas do casamento de padrões, ou seja, elimina (transforma) os padrões que eventualmente possam estar presentes nas expressões. Aquelas expressões que não apresentam padrões, são apenas copiadas para que se possa realizar o *lambada-lifting* na etapa seguinte. Maiores detalhes de como é feito um casamento

[1] Do inglês *binding*.

de padrões ideal e sobre a especificação da etapa podem ser vistos na Seção 4.2.2.

5.1.1 Representação Interna de Programas $\mathcal{LAMB}$

Nesta seção pode-se ver como é feita a representação das expressões $\mathcal{LAMB}$ na memória do computador. Em qualquer implementação baseada em redução de grafos, as expressões a serem avaliadas são mantidas na memória na forma de árvore sintática. Normalmente, as folhas dessas árvores são identificadores, funções *built-in* ou valores constantes. Cada nodo da árvore contém um campo chamado *tag* que indica a natureza da sub-árvore que se inicia a partir dele.

Para a aplicação de uma função f a um argumento x, temos, por exemplo, a seguinte árvore sintática:

```
    @
   / \
  f   x
```

No caso ilustrado nesta figura o símbolo '@' é o *tag* do nodo, indicando, nesse caso, uma aplicação de função.

Na realidade, programas em $\mathcal{LAMB}$ são representados internamente por meio de três tipos de estruturas: representação de expressões, representação de padrões e representação de pares formados por lados esquerdo e direito de uma definição LET/DEF.

1. Representação de Expressões

A estrutura que representa expressões é uma lista de nodos os quais são denominados **elementos de código.** Cada elemento de código é, na realidade, um token. No caso do token ser um identificador, uma constante inteira ou uma constante string (quotation), faz-se necessária uma informação adicional além do tipo do token: o valor dessa constante ou desse identificador. Por exemplo, se o token for um identificador, deve ser informado qual é o identificador. Sendo, por exemplo, o identificador 'x', deve-se ter algo como (_IDE,x) como pode ser visualizada na Figura 5.1. Nessa mesma figura temos a constante TT. Uma última informação é necessária para compor o nodo: um campo para determinar qual é o próximo elemento do código.

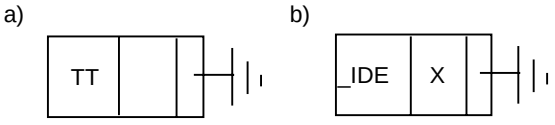

Figura 5.1: Representação de Elementos de Código de Expressões $\mathcal{LAMB}$.

A Figura 5.1 sugere a seguinte representação da linguagem C para elementos de código:

```
typedef struct strOther {
        } tOther;
typedef struct strInteiro {
          char* valor;
        } tInteiro;
typedef struct strString {
          char* valor;
        } tString;
typedef struct strIdentificador {
          char* valor;
        } tIdentificador;
typedef struct strCodeElement {
          int         tipo;
          union {
            tInteiro       inteiro;
            tString        string;
            tIdentificador identificador;
            tOther         other;
          } info;
struct strCodeElement *elo;
        } tCodeElement;
```

Um código para expressões é uma lista desses elementos de código. Esse código é representado por dois apontadores: um para o início e outro para o fim da lista dos elementos de código. Pode-se visualizar um código na Figura 5.2.

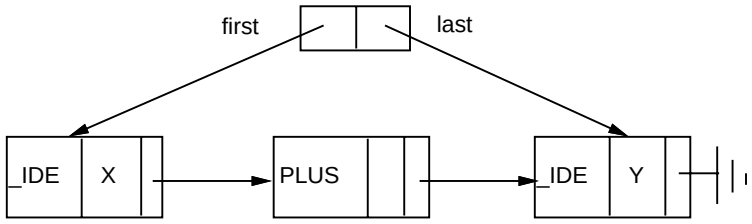

Figura 5.2: Representação do Código da Expressão $\mathcal{LAMB}$ "x PLUS y".

A representação para a estrutura do código em linguagem C é a seguinte:

```
typedef struct strCode {
          tCodeElement *first, *last;
        } tCode;
```

2. Representação de Padrões

Os padrões são representados como árvores cujos nodos possuem um número arbitrário de filhos. Apesar disso, para facilitar a implementação, essas árvores são representadas como árvores binárias. Ou seja, cada nodo é representado por uma célula de memória que contém um *tag* que indica o tipo da célula: tuplas, células PRE ou NODE, constantes inteiras, booleanas, etc. Essa célula possui ainda mais três campos: um para especificar o valor de um identificador ou constante, um para indicar o filho e um último campo para indicar o irmão do nodo. A Figura 5.3 mostra a estrutura de uma célula "padrão".

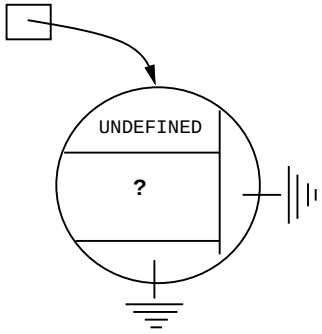

Figura 5.3: Representação de um Nodo do Padrão "UNDEFINED".

De acordo com a Figura 5.3, cada nodo da árvore binária é representado como uma estrutura da linguagem C:

```
typedef struct strPatElement {
          int         tipo;
          union {
            tInteiro       inteiro;
            tString        string;
            tIdentificador identificador;
            int            token;
          }           info;
          struct strPatElement *son, *brother;
        } tPatElement;
```

Uma observação importante a ser feita é que os dados estruturados, como as tuplas, são construídos por funções construtoras como PRE e NODE. Esses dados consistem em um agregado de valores que é representado de maneira similar às outras construções existentes na linguagem. Em nossa implementação as células são de tamanho fixo, o que obriga que os dados estruturados sejam implementados como uma coleção de células encadeadas, como pode ser visto na Figura 5.4.

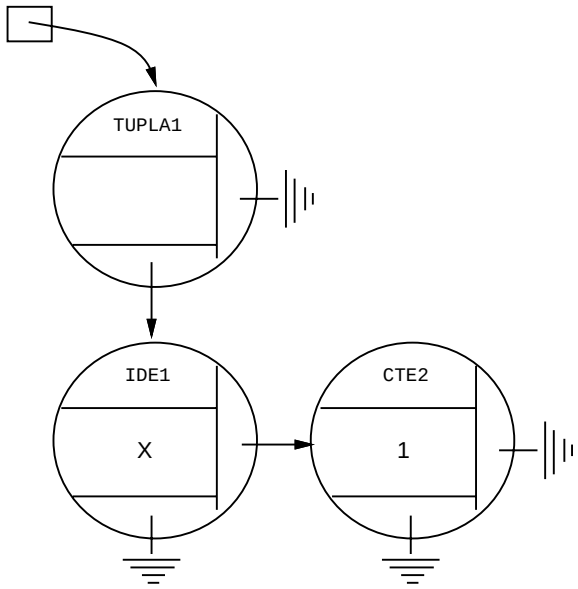

Figura 5.4: Representação Interna do Padrão Tupla "(x,1)".

As representações para todos os tipos de padrões podem ser encontradas no Apêndice D.

3. Representação de Pares de Listas de Definições

Antes de gerar código para uma lista de definições LET/DEF, há uma coleta de todos os pares de definição envolvidos na lista. Na prática, quando há uma lista de definições como a seguinte

```
LET x = 1 ALSO y = 2 ALSO z = 3 ALSO ? = 9 ALSO 3 = 3 IN x PLUS y PLUS z
```

obtém-se a lista de pares < (x,1), (y,2), (z,3), (?,3 NE 3) > . Esses pares são constituídos assim: o primeiro elemento ou será um identificador, no caso do lado esquerdo da definição ser qualquer coisa diferente de uma constante, ou será a constante UNDEFINED no caso do lado esquerdo da definição ser uma constante. O segundo elemento do par será a expressão do lado direito da definição que estiver sendo verificada caso o respectivo lado esquerdo seja qualquer padrão diferente de constante ou será uma expressão que testa se a expressão do lado direito casa com o padrão do lado esquerdo da definição. Alguns casos podem ser vistos no exemplo dado e a abordagem mais genérica pode ser verificada no Capítulo 4, onde está toda a especificação da tradução. O padrão UNDEFINED é eliminado nesses casos, pois não faz sentido expressões da forma LET ? = $\cdots$, apesar da gramática o permitir.

Após coletados todos os pares da lista de definições, gera-se o código final para o LET ou DEF correspondente. No caso do primeiro elemento do par ser a constante UNDEFINED,

o código gerado é um teste condicional. Caso o teste seja satisfeito, a resposta é UNDEFINED. Caso contrário, retorna falso e a expressão segue avaliando. No caso do primeiro elemento do par ser um identificador, é gerada a seqüência: o operador LET/DEF correspondente seguido pelo primeiro elemento do par, o token EQUAL, o segundo elemento do par e o token IN. Portanto, o último exemplo ficaria traduzido assim:

```
LET x = 1 IN LET y = 2 IN LET z = 3 IN 3 NE 3 -> ?, x PLUS y PLUS z
```

A Figura 5.5 ilustra a representação para uma lista de pares.

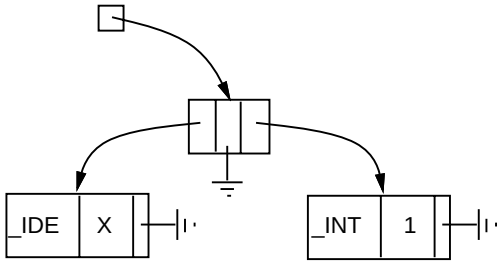

Figura 5.5: Representação Interna do Par "(x,1)".

A representação de um par, em linguagem C, está de acordo com a Figura 5.5:

```
typedef struct strPair {
          tCode *fst, *snd;
          struct strPair *next;
        } tPair;
```

5.2 Estruturas de Dados utilizadas na Tradução de Programas *LAMBASIC* para *SUPER*

Após ter concluído a primeira etapa de compilação, o objetivo agora é transformar a expressão gerada em uma coleção de supercombinadores mais uma expressão a ser avaliada.

A coleção de supercombinadores gerada é acrescida de supercombinadores auxiliares, conforme especificado na Seção 4.1. Neste momento da tradução, o estilo de compilação adotado para a etapa anterior foi mantido, ou seja: a expressão traduzida é uma seqüência de elementos de código, que são tokens da linguagem. Mas o código gerado agora é uma tripla formada por: (1) definições de supercombinadores e funções C, (2) expressão a ser avaliada e, (3) lista de variáveis livres. Todo o formalismo adotado foi exposto na Seção 4.3.

Além das expressões, as definições de supercombinadores também exigem uma lista de elementos de código, o que faz com que a mesma estrutura para expressões utilizada na etapa anterior seja aproveitada. Já a lista de variáveis livres tem uma abordagem diferente que será mostrada logo adiante.

A Figura 5.6 ilustra a estrutura utilizada para expressões e supercombinadores.

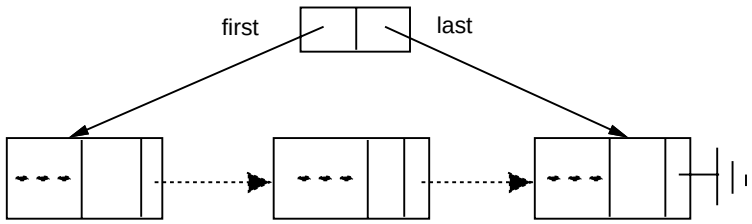

Figura 5.6: Representação Interna para Expressões e Definições de Supercombinadores.

O código C para a Figura 5.6 é:

```
typedef struct strOther {
        } tOther;

typedef struct strInteiro {
          char* valor;
        } tInteiro;

typedef struct strString {
          char* valor;
        } tString;

typedef struct strIdentificador {
          char* valor;
        } tIdentificador;

typedef struct strCodeElement {
          int         tipo;
          union {
            tInteiro       inteiro;
            tString        string;
            tIdentificador identificador;
            tOther         other;
          } info;
          struct strCodeElement *elo;
```

```
        } tCodeElement;

typedef struct strCode {
          tCodeElement *first, *last;
        } tCode;

typedef tCode tSuper;

typedef tCode tExpr;
```

O terceiro componente da tripla obtida na tradução de código $\mathcal{LAMBASIC}$ para $\mathcal{SUPER}$, a lista de variáveis livres, é composta por elementos encadeados. Cada elemento desse é formado por dois campos: o primeiro é um índice que informa qual é a entrada na Tabela de Símbolos, o segundo é um apontador que informa qual é o próximo identificador livre presente na lista. A Figura 5.7 ilustra esta estrutura. Ela mostra que os identificadores das posições 1, 2 e 3 da Tabela de Símbolos estão livres na expressão.

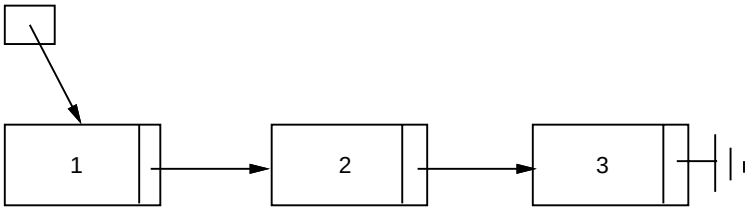

Figura 5.7: Representação Interna para Elemento de Lista de Variáveis Livres.

O código C correspondente é:

```
typedef struct strFreeElement {
          int  entry;
          struct strFreeElement *next;
        } tFreeElement;
```

Para finalizar, é apresentada a representação interna da tripla gerada na produção de um supercombinador. Ao terminar a tradução, o código $\mathcal{SUPER}$ é constituído pelas definições de supercombinadores, a expressão que será avaliada, fazendo uso possivelmente dos supercombinadores gerados e/ou dos auxiliares, e uma lista de variáveis livres que estará provavelmente vazia. A Figura 5.8 ilustra esta tripla e o código C a seguir mostra sua representação:

```
typedef struct strCodeSuper {
          tSuper *super;
          tExpr *expr;
          tFreeElement *freeQueue;
        } tCodeSuper;
```

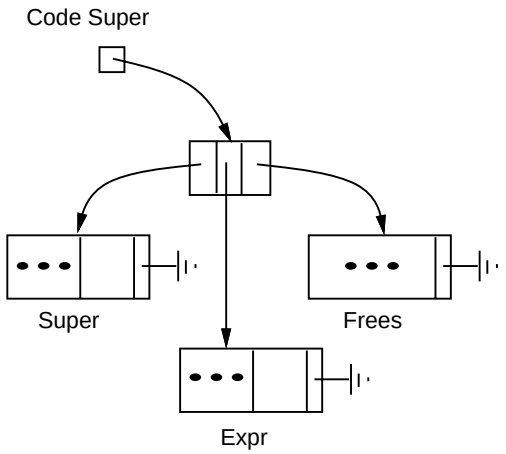

Figura 5.8: Representação Interna para o Código de Supercombinadores.

A representação interna para *definition* e *defn* é um pouco diferente da tripla apresentada anteriormente. Essas construções necessitam de um campo a mais: a entrada na Tabela de Símbolos correspondente ao identificador que está sendo ligado pela definição *defn*. Isso justifica-se porque pode ser necessário fazer alguma alteração nos seus atributos na Tabela de Símbolos.

As estruturas para *definition* e para o código dos supercombinadores são bem semelhantes, dispensando maiores comentários. A Figura 5.9 e o código C a seguir ilustram a idéia. O nome ECS faz uma alusão a *Extended Code Super*.

```
typedef struct strECS {
          tCodeSuper *codeSuper;
          tEntryTS *entry;
        } tECS;
```

5.3 Conclusão

Após completar as etapas 1 e 2, está gerado o código $\mathcal{SUPER}$ de supercombinadores a partir de um código $\mathcal{LAMB}$ original. Ressaltamos que a presença de padrões na linguagem

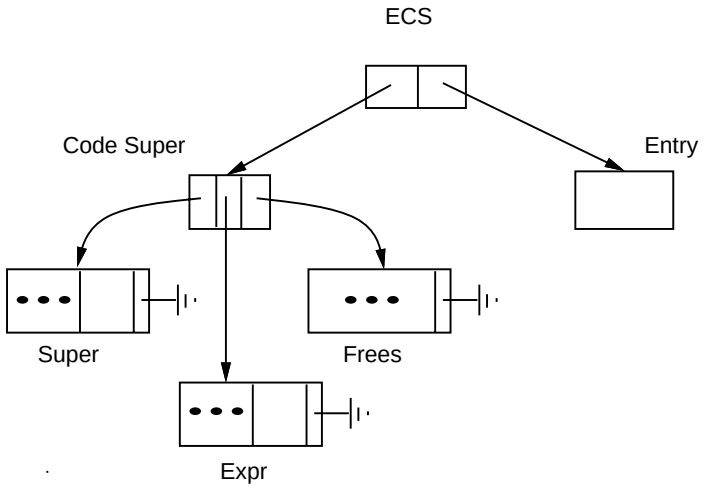

Figura 5.9: Representação Interna para o Código de Supercombinadores Estendido (ECS).

fonte inviabiliza a geração do código em apenas um passo, conforme discutido e justificado no Capítulo 4.

Outro ponto importante que deve ser levantado é a natureza dos atributos dos símbolos das gramáticas de $\mathcal{LAMB}$ e de $\mathcal{LAMBASIC}$. Eles são basicamente códigos, como foi exposto nas Seções 5.1 e 5.2. Como é gerado código binário a partir desses atributos, ao final do processo há uma conversão deste código para texto para que o programa $\mathcal{SUPER}$ seja lido pelo compilador de Maia [Maia, 1994].

Capítulo 6

Resultados Obtidos

Neste capítulo são apresentados alguns resultados que foram obtidos executando-se as duas fases do compilador de $\mathcal{LAMB}$. São apresentados os programas $\mathcal{LAMB}$, $\mathcal{LAMBASIC}$ e $\mathcal{SUPER}$. Os programas $\mathcal{LAMB}$ são arquivos constituídos de tokens no formato binário e foram gerados pelo *Front-End* [Oliveira, 1998], enquanto os programas $\mathcal{SUPER}$ são arquivos de lexemas no formato texto e são lidos pelo *Back-End* [Maia, 1994]. O Apêndice E mostra como utilizar o programa.

Para o leitor mais interessado em exemplos detalhados, foi anexado o Apêndice F.

Mais uma observação é importante: são apresentados apenas alguns programas exemplos, mas a variedade e quantidade de expressões testadas nesses programas pode ser considerada satisfatória.

Exemplo 1: operadores binários e constantes

```
-- LAMB

 LET TesteScript  =
  LET ndom1   = 11
  IN
  LET ndom2   = ndom1   PLUS 11
  IN
  LET ndom3   = ndom2   MULT 11
  IN  <  >


-- LAMBASIC

 LET TesteScript  =
  LET ndom1   = 11
  IN
  LET ndom2   = ndom1   PLUS 11
  IN
  LET ndom3   = ndom2   MULT 11
  IN  <  >


-- SUPER

$S35 = 11

$S36 = PLUS $S35 11

$S37 = MULT $S36 11

$MOD_LAMB = ( )
```

Exemplo 2: abstração lambda e constantes

```
-- LAMB

 LET TesteScript  =
  LET ndom1   = 11
  ALSO t1   = "FF"
  ALSO f1   =   LAM   ( x   )   . x   EQ 0  -> 1 , 2
  IN  <  >

-- LAMBASIC

 LET TesteScript  =
  LET ndom1   = 11
  IN
  LET t1   = "FF"
  IN
  LET f1   =   (   LAM nn24   .   (   LAM x   . x   EQ 0   -> 1   , 2   )
   ( nn24   EL 1   )   )
  IN  <  >


-- SUPER


$S35 = 11

$S36 = "FF"

$S38 x = IF EQ x 0 1 2

$S39 nn24 = AP ( $S38 ) ( EL nn24 1 )

$S37 = ( $S39 )

$MOD_LAMB = ( )
```

Exemplo 3: operador IS

```
-- LAMB

 LET TesteFive  =
  LET n1   = 3 PLUS 7
  ALSO t1   = n   IS 10
  IN  <  >

-- LAMBASIC


 LET TesteFive  =
  LET n   = 3   PLUS 7
  IN
  LET t1   =   ( n   EQ 10   )   ->   TT   ,   FF
  IN  <  >


-- SUPER


$S35 = PLUS 3 7

$S36 = IF ( EQ $S35 10 ) TT FF

$MOD_LAMB = ( )
```

Exemplo 4: aplicação de função

```
-- LAMB

 LET TesteScript  =
  LET f   =   LAM a   . a   PLUS 1
  IN
  LET b   = f 3
  IN  <  >


-- LAMBASIC

 LET TesteScript  =
  LET f   =   (   LAM nn22   .   (   LAM a   . a   PLUS 1   )   ( nn22   )   )
  IN
  LET b   = f 3
  IN  <  >


-- SUPER


$S36 a = PLUS a 1

$S37 nn22 = AP ( $S36 ) ( nn22 )

$S35 = ( $S37 )

$S38 = AP $S35 3

$MOD_LAMB = (  )
```

Capítulo 7

Conclusão

O trabalho desenvolvido completa o ciclo de criação de uma nova linguagem de programação: $\mathcal{SCRIPT}$. Embora as linguagens funcionais ainda apresentem um desempenho de execução muito baixo em relação às linguagens tradicionais dos paradigmas imperativo e Orientado por Objetos, escrever programas em $\mathcal{SCRIPT}$ é uma tarefa simples e amigável, o que pode tornar seu uso bastante comum.

Por se enquadrar dentro do paradigma funcional, $\mathcal{SCRIPT}$ possui uma maneira intuitiva de programar, já que as instruções são especificações matemáticas tradicionais. Além da rapidez com que se pode construir programas de propósito geral, $\mathcal{SCRIPT}$ apresenta um grande diferencial: apresenta facilidades para especificação denotacional de linguagens de programação.

A tradução de $\mathcal{LAMB}$ para $\mathcal{SUPER}$ não pode ser configurada apenas como mais uma etapa do compilador para $\mathcal{SCRIPT}$. As contribuições deste trabalho compreendem ainda o estudo e desenvolvimento de técnicas para compilação de linguagens funcionais, apresentando e implementando temas importantes: há um compilador de casamento de padrões na Seção 5.1, simples e eficiente acoplado à implementação do algoritmo clássico de *lambda-lifting,* descrito na Seção 5.2, proposto por Peyton Jones [Jones, 1987]. Apesar da qualidade e completeza do compilador construído, há ainda pontos susceptíveis a melhorias e trabalhos futuros que devem ser levantados. A Seção 7.1 apresenta esses pontos.

7.1 Trabalhos Futuros

Um ponto importante para trabalho futuro é a inserção de otimizações na geração do código $\mathcal{SUPER}$. Algumas otimizações simples foram feitas, como por exemplo a ordenação de variáveis livres pelo seu nível léxico, nível de escopo, para a colocação delas como parâmetro de supercombinadores (esse tipo de otimização está ausente, por exemplo, no trabalho de Peyton Jones de David Lester [Jones and Lester, 1994], salientado na Seção 3.5). Mas outras ainda podem ser realizadas. Exemplos importantes seriam: (1) efetuar η-transformações nas definições de supercombinadores e, como etapa seguinte, (2) eliminar

definições redundantes de supercombinadores. Para ilustrar, considere por exemplo se $X x = $Y x, então $X = $Y , representando uma *eta*-redução. Se $X = $Y, então não há necessidade de existir a definição do supercombinador $X ou $Y. Qual definição será mantida e quais as transformações feitas em outras partes do código dependerão da política de otimização adotada.

Um segundo aspecto muito importante para implementação foi levantado por Maia em seu trabalho [Maia, 1994]: a anexação de um *garbage collector* [Jones, 1987]. O *garbage collector* melhora o desempenho na utilização da memória da máquina conforme as reduções vão sendo processadas. Veja Seção D.2 para mais informações sobre um módulo para *garbage collection.*

Para escrever programas em $\mathcal{SCRIPT}$ e obter um programa C automaticamente, o usuário executa cada etapa do compilador de maneira transparente, via um "shell script" *sc*[1] que aplica o *Front-End* [Oliveira, 1998] no código fonte $\mathcal{SCRIPT}$ para obter o código intermediário $\mathcal{LAMB}$; aplica o resultado obtido ao algoritmo de *lambda-lifting* para gerar código $\mathcal{SUPER}$, e; para finalizar, executa o *Back-End* [Maia, 1994] para gerar código de Máquina-G [Jones, 1987] a partir dos supercombinadores produzidos pela segunda etapa do compilador. Para que o trabalho se torne mais interessante, pode-se agrupar as fases em um ambiente de programação com editor de texto, *help on line* e outras ferramentas mais.

[1] Verificar Apêndice E.

Bibliografia

[Aho et al., 1986] Aho, A. V., Ullman, J. D., and Sethi, R. S. (1986). *Compilers, Principles, Techniques and Tools*. Addison Wesley Publishing Company Co.

[Augustsson, 1984] Augustsson, L. (1984). *A compiler for Lazy ML*. In *ACM Symposium on Lisp and Functional Programming*, pages 218–227.

[Augustsson and Johnsson, 1989] Augustsson, L. and Johnsson, T. (1989). *The Chalmers Lazy-ML Compiler*. In *Computer Journal*, volume 32, pages 127–141.

[Berry and Schwartz, 1979] Berry, D. M. and Schwartz, R. L. (1979). *Type Equivalence in Strongly Typed Languages: One More Look*. In *ACM SIGPLAN Notices*, pages 158–166.

[Bigonha, 1981] Bigonha, R. S. (1981). *A Denotational Semantics Implementation System*. PhD thesis, University of California, Los Angeles - 428 pages, Departament of Computer Science and Engineering.

[Bigonha, 1997] Bigonha, R. S. (1997). *The Revised Report on the SCRIPT Language for Denotational Semantics*. Technical Report RT016/97, Departamento de Ciência da Computação.

[Cardelli, 1983] Cardelli, L. (1983). *The Functional Abstract Machine*. In *Polimorphism*, volume 1.

[Cardelli, 1984] Cardelli, L. (1984). *Compiling a Functional Language*. In *ACM Symposium on Lisp an Functional Programming*, pages 208–217.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). *On Understanding Types, Data Abstraction and Polymorphism*. In *ACM Computing Surveys*, number 17, pages 471–522.

[Gordon, 1979] Gordon, M. J. C. (1979). *The Denotational Description of Programming Languages - An Introduction*. Springer Verlag.

[Hermano, 1996] Hermano, P. M. (1996). *An Overview of Action Semantics. I Simpósio de Linguagens de Programação*, pages 277–304.

[Hindley and Seldin, 1986] Hindley, J. R. and Seldin, J. P. (1986). *Introduction to Combinators and λ-Calculus*. Cambridge University Press.

[Johnson, 1978] Johnson, S. C. (1978). YACC: Yet Another Compiler-Compiler.

[Johnsson, 1984] Johnsson, T. (1984). *Efficient compilation of Lazy Evaluation*. In *ACM Conference on Compiler Construction*, pages 58–69.

[Jones, 1987] Jones, S. L. P. (1987). *The Implementation of Functinal Programming Languages*. Prentice-Hall International Series in Computer Science.

[Jones and Lester, 1994] Jones, S. L. P. and Lester, D. R. (1994). Implementing Functional Languages: a tutorial.

[Júnior, 1993] Júnior, J. L. S. (1993). *Linguagem de Definição e Geração de Analisadores Sintáticos em Semântica Denotacional Legível* . Master's thesis, Universidade Federal de Minas Gerais.

[Kernighan and Ritchie, 1988] Kernighan, B. and Ritchie, D. (1988). *The C Programming Language*. Prentice Hall.

[Knuth, 1984] Knuth, D. E. (1984). *Literate Programming*. In *The Computer Journal*, volume 27, pages 97–111.

[Lamport, 1986] Lamport, L. (1986). LaTeX - *A Document Preparation System*. Addison Wesley Publishing Company Co.

[Landin, 1964] Landin, P. J. (1964). *The Mechanical Evaluation of Expressions*. In *The Computer Journal*, number 6, pages 308–320.

[Lins, 1987] Lins, R. D. (1987). *Categorical multi-combinators*. Springer Verlag.

[Maia, 1994] Maia, M. A. (1994). *Implementação Eficiente de uma Linguagem para Definição de Semântica*. Master's thesis, Universidade Federal de Minas Gerais.

[Meyer, 1988] Meyer, B. (1988). *Object-oriented Software Construction*. Prentice-Hall International Series in Computer Science.

[Mosses, 1975] Mosses, P. D. (1975). *Mathematical Semantics and Compiler Generation*. PhD thesis, Oxford University, Computing Laboratory.

[Mosses, 1978] Mosses, P. D. (1978). *A Compiler-Generator System using Denotational Semantics*. Technical report, University of Aarhus, Denmark.

[Oliveira, 1998] Oliveira, F. F. (1998). *Compilação Eficiente de uma Linguagem Funcional, Orientada por Objetos, para Definição Semântica Denotacional* . Master's thesis, Universidade Federal de Minas Gerais.

[Rodrigues, 1993] Rodrigues, W. A. (1993). *Compilação e Otimização de uma Linguagem para Definição Denotacional de Semântica* . Master's thesis, Universidade Federal de Minas Gerais.

[Sussman, 1982] Sussman, G. J. (1982). *LISP, Programming and Implementation.* Cambridge University Press, London.

[Thomas and Yates, 1988] Thomas, D. R. and Yates, J. (1988). *UNIX Total: Guia do Usuário.* Editora McGraw-Hill Ltda.

[Thompson, 1996] Thompson, S. (1996). *Haskell. The Craft of Functional Programming.* Addison-Wesley.

[Turner, 1986] Turner, D. (1986). *An Overview of Miranda.* In *ACM SIGPLAN Notices*, volume 21, pages 158–166.

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

[Wakeling, 1998] Wakeling, D. (1998). *The Dynamic Compilation of Lazy Functional Programs. Functional Programming*, 8(1):61–81.

[Wirth, 1976] Wirth, N. (1976). *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, New Jersey.

Apêndice A

A Sintaxe de $\mathcal{LAMB}$

Neste apêndice é apresentada a gramática abstrata de $\mathcal{LAMB}$ na notação BNF padrão. As ambigüidades em aplicações de funções foram retiradas. É bom observar que nesta gramática ocorre a existência de padrões.

```
mod_lamb    ::= "LET" ide "=" exp

exp         ::= "LAM"  pat  "."  exp
              | defn_list  "IN"  exp
              | exp  "->"  exp  ","  exp
              | exp op_di exp
              | exp  "IS"  pat
              | op_mono exp
              | application
              | "("  tuple_list  ")"
              | "<"  tuple_list  ">"
              | ide
              | atom

application ::= appl_a
              | appl_b
              | appl_c
              | appl_d

appl_a      ::= exp exp
              | application exp

appl_b      ::= exp ";" exp
              | exp ";" application
```

```
appl_c      ::= exp CIRC exp
              | application CIRC exp

appl_d      ::= exp STAR exp
              | application STAR exp

defn_list   ::= defn_list let_list
              | defn_list def_list
              | let_list
              | def_list

let_list    ::= "LET" defn_a

def_list    ::= "DEF" defn_b

defn_a      ::= defn_a "ALSO" defn
              | defn

defn_b      ::= defn_b "WITH" defn
              | defn

defn        ::= pat  "="  exp

tuple_list  ::= exp_list
              | ε


exp_list    ::= exp_list "," exp
              | exp

atom        ::= n
              | q
              | "TT"
              | "FF"
              | "?"

pat         ::= "?"
              | "<"  ">"
              | ide
              | "SPECIAL"  ide  ":"  type
              | ide  ":"  type
              | pat  "PRE"  pat
              | pat  "NODE"  pat
```

```
              | tuple_pat
              | atom
              | "LAM"  "?"   "."   "?"
              | "NUMBER"  ide
              | "NUMBER"  "?"
              | "TRUTH"  ide
              | "TRUTH"  "?"
              | "QUOTE"  ide
              | "QUOTE"  "?"
              | "VAL"  ide
              | "VAL"  "?"

tuple_pat   ::= "("  pat_list  ")"
              | ide  "["  n  "]"  "EXT"  tuple_pat
              | tuple_pat  "EXT"  tuple_pat

pat_list    ::= pat_list_a
              | ℰ


pat_list_a  ::= pat_list_a "," pat
              | pat

type        ::= "NUM"
              | "QUOTE"
              | "BOOL"

op_mono     ::= "CONC"  | "HEAD" | "NOT"  | "NUMBER"
              | "QUOTE" | "SIZE" | "TAIL" | "TRUTH"

op_di       ::= "AND" | "AUG"   | "CAT"  | "DIV" | "EL"
              | "EQ"  | "EXT"   | "GE"   | "GR"  | "LE"
              | "LS"  | "MINUS" | "MULT" | "NE"  | "NODE"
              | "OR"  | "PLUS"  | "PRE"  | "REM"

op_reps     ::= op_reps op_rep
              | ℰ


op_rep      ::= "*"  |  "+"

ide            ::= letter letter_dashes digits primes op_reps

letter_dashes  ::= letter_dashes letter_dash
```

```
                | ε

digits          ::= digits digit
                | ε

primes  ::= primes prime
          | ε

letter_dash ::= letter | "-"

n           ::= digit+

q           ::= symbol*

prime       ::= "'"

symbol      =/= "\""

letter      === "a" .. "z"  |   "A" .."Z"

digit       === "0" .. "9"
```

Apêndice B

A Sintaxe de $\mathcal{LAMBASIC}$

Neste apêndice é apresentada a gramática de $\mathcal{LAMBASIC}$, um cálculo lambda estendido sem padrões. Assim como a sintaxe do Apêndice A, esta é uma gramática abstrata e encontra-se na notação BNF padrão. O nome $\mathcal{LAMBASIC}$ faz uma alusão a Cálculo Lambda Básico, sem padrões.

```
mod_lambasic ::= "LET" ide "=" exp

exp          ::= "LAM"  ide_b  "."  exp
               | definition  "IN"  exp
               | exp  "->"  exp  ","  exp
               | exp op_di exp
               | op_mono exp
               | application
               | "("  tuple_list  ")"
               | "<"  tuple_list  ">"
               | ide
               | atom

application ::= exp exp
               | application exp

definition  ::= let_definition
               | def_definition

let_definition  ::= "LET" defn
                  | "LET" "SPECIAL" defn

def_definition  ::= "DEF" defn
                  | "DEF" "SPECIAL" defn
```

```
defn        ::= ide_b  "="  exp

tuple_list  ::= exp_list
              | ℰ


exp_list    ::= exp_list "," exp
              | exp

atom        ::= n
              | q
              | "TT"
              | "FF"
              | "?"

ide_b       ::= ide
              | ide  ":"  type

type        ::= "NUM"
              | "QUOTE"
              | "BOOL"

op_mono     ::= "CONC"  | "HEAD" | "NOT"  | "NUMBER"
              | "QUOTE" | "SIZE" | "TAIL" | "TRUTH"

op_di       ::= "AND" | "AUG"   | "CAT"  | "DIV" | "EL"
              | "EQ"  | "EXT"   | "GE"   | "GR"  | "LE"
              | "LS"  | "MINUS" | "MULT" | "NE"  | "NODE"
              | "OR"  | "PLUS"  | "PRE"  | "REM"

op_reps     ::= op_reps op_rep
              | ℰ


op_rep      ::= "*"  |  "+"

ide           ::= letter letter_dashes digits primes op_reps

letter_dashes  ::= letter_dashes letter_dash
                 | ℰ


digits         ::= digits digit
```

```
                | ε

primes  ::= primes prime
          | ε


letter_dash ::= letter | "-"

n           ::= digit+

q           ::= symbol*

prime       ::= "'"

symbol      =/= "\""

letter      === "a" .. "z"  |   "A" .."Z"

digit       === "0" .. "9"
```

Apêndice C

A Sintaxe da Linguagem $\mathcal{SUPER}$ de Supercombinadores

A gramática de supercombinadores encontrada neste apêndice é a mesma proposta em [Maia, 1994].

```
Prog              ::= ProgExe | ProgObj

ProgExe           ::= "PROG" ProgObj

ProgObj           ::= DefSuper ProgObj
                    | "CFUN" DefCfun ProgObj
                    | ε


DefCfun           ::= IdeType IdeCfun ListaTypedParams "=" Corpo

DefSuper          ::= IdeSuper ListaParams "=" Corpo

ListaTypedParams  ::= IdeType Param ListaTypedParams
                    | ε


ListaParams       ::= Param ListaParams
                    | ε


Corpo             ::= Inteiro
                    | Booleano
                    | Tupla
                    | "IF" Corpo Corpo Corpo
```

```
                    | "AP" Corpo Corpo
                    | Op_Bin Corpo Corpo
                    | Op_Un Corpo
                    | "LET" ListDefs "IN" Corpo
                    | "DEF" ListDefs "IN" Corpo
                    | Ide

Tupla               ::= "(" RestoTupla ")"

RestoTupla          ::= Corpo RestoTupla
                    | ℰ

ListDefs            ::= Defn ListDefs
                    | ℰ

Defn                ::= Ide "=" Corpo

Ide                 ::= <LEXEME_ID>

IdeSuper            ::= <LEXEME_ID>

IdeCfun             ::= <LEXEME_ID>

IdeType             ::= "int"  |  "string"

Param               ::= <LEXEME_ID>

Booleano            ::= "TT"  |  "FF"

Inteiro             ::= <LEXEME_INT>

Op_Un               ::= "CONC"  |  "NOT"  | "NUMBER"
                    | "QUOTE" | "SIZE"

Op_Bin              ::= "AND" | "AUG"   | "CAT"  | "DIV" | "EL"
                    | "EQ"  | "EXT"   | "GE"   | "GR"  | "LE"
                    | "LS"  | "MINUS" | "MULT" | "NE"  | "NODE"
                    | "OR"  | "PLUS"  | "PRE"  | "REM"
```

Apêndice D

Representação Interna de Programas $\mathcal{LAMB}$

Neste apêndice mostramos como é feita a representação das expressões lambda $\mathcal{LAMB}$ na memória do computador. Há, na realidade, três estruturas essenciais e bastante diferentes para representação das expressões $\mathcal{LAMB}$, conforme dito no Capítulo 5: uma para expressões, outra para padrões e uma terceira para pares (l,r): lados esquerdo e direito de uma definição LET/DEF. A Seção D.1 apresenta figuras para um melhor entendimento das representações utilizadas na implementação dos padrões.

A representação interna para expressões e pares utilizada na Etapa 1 da compilação é bem simples, e pode ser encontrada no Capítulo 5.

D.1 Representação Interna para Padrões

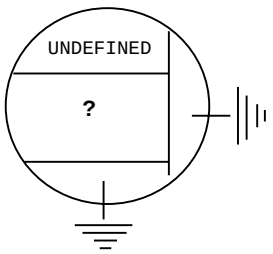

Figura D.1: Representação do Padrão "UNDEFINED": '?'.

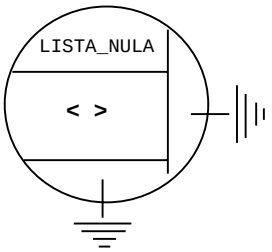

Figura D.2: Representação do Padrão "LISTA_NULA": '< >'.

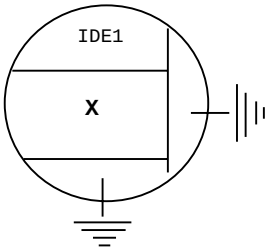

Figura D.3: Representação do Padrão "IDE1": 'X'.

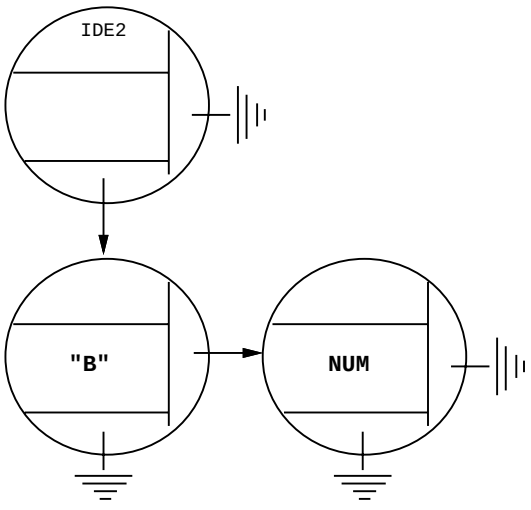

Figura D.4: Representação do Padrão "IDE2": 'SPECIAL B: NUM'.

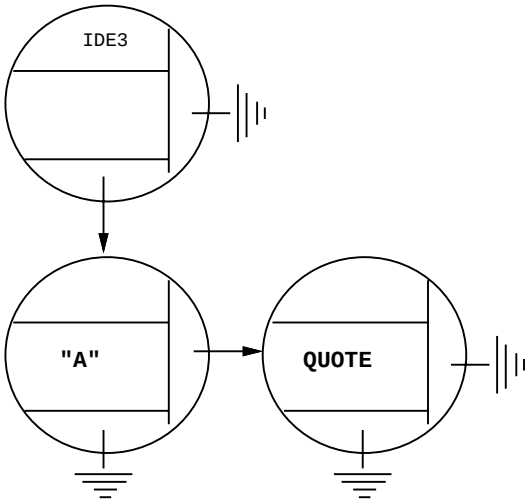

Figura D.5: Representação do Padrão "IDE3": 'A: QUOTE'.

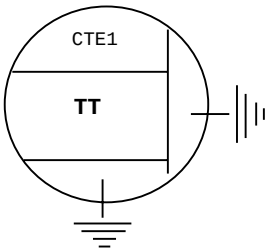

Figura D.6: Representação do Padrão "CTE1": 'TT'.

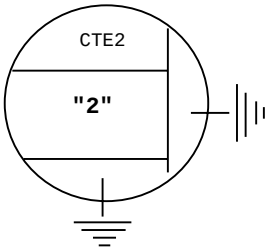

Figura D.7: Representação do Padrão "CTE2":'2'.

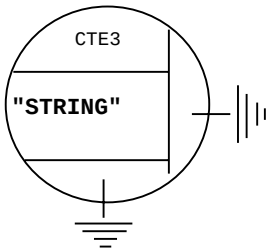

Figura D.8: Representação do Padrão "CTE3": 'STRING'.

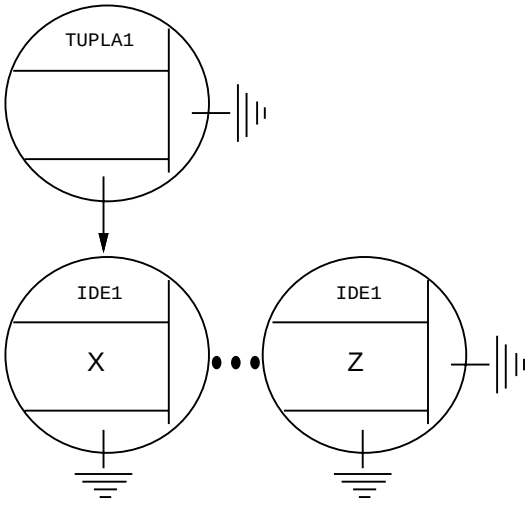

Figura D.9: Representação do Padrão "TUPLA1": '(X,$\cdots$,Z)'.

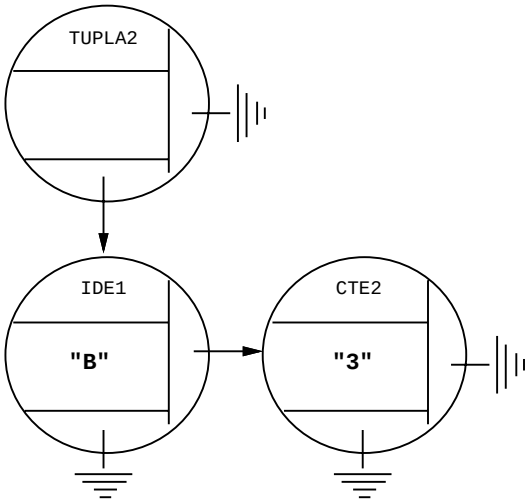

Figura D.10: Representação do Padrão "TUPLA2":'B[3]'.

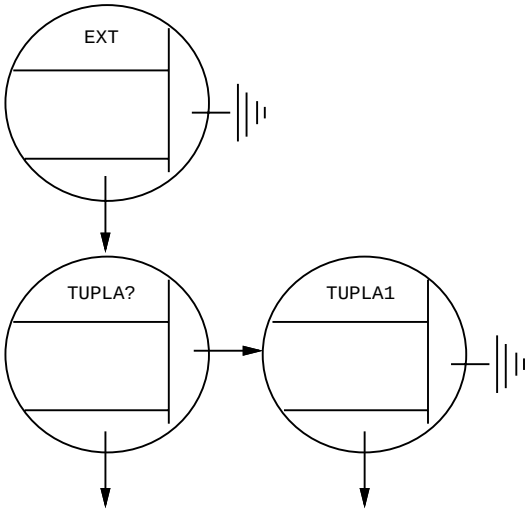

Figura D.11: Representação do Padrão "EXT": (TUPLA1 ou TUPLA2) EXT TUPLA1.

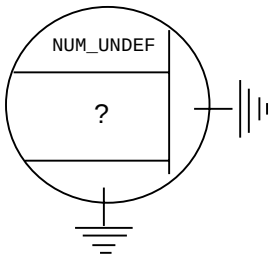

Figura D.12: Representação do Padrão "NUM_UNDEF":'NUMBER ?'.

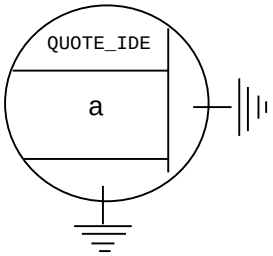

Figura D.13: Representação do Padrão "QUOTE_IDE":'QUOTE a'.

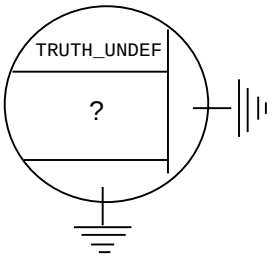

Figura D.14: Representação do Padrão "TRUTH_UNDEF":'TRUHT ?'.

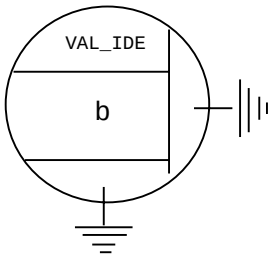

Figura D.15: Representação do Padrão "VAL_IDE": 'VAL b'.

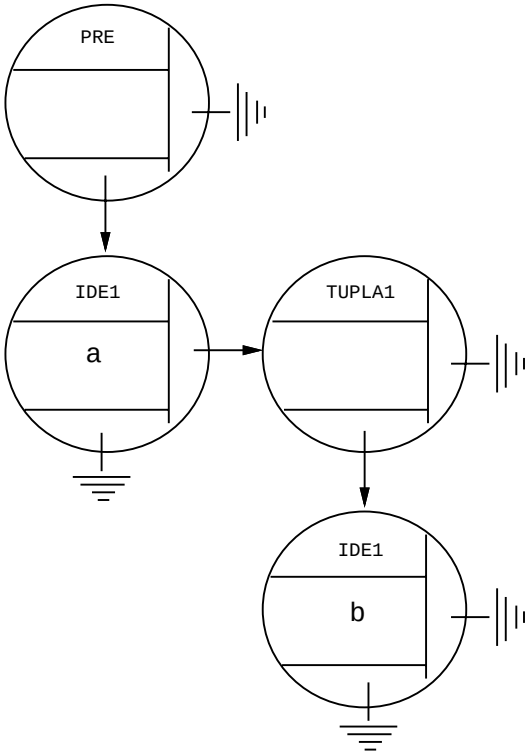

Figura D.16: Representação do Padrão "PRE": 'a PRE b'.

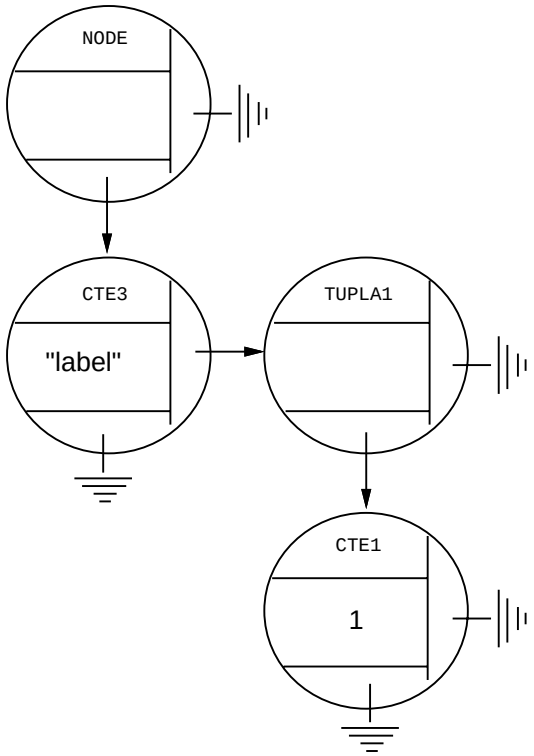

Figura D.17: Representação do Padrão “NODE”: ‘ “label” NODE (1)’.

D.2 Gerenciamento de Memória e Coleta de Lixo

Enquanto a tradução se realiza, vários trechos de programa precisam ser construídos, ou seja, é necessário serem alocadas novas células a todo momento. Além disso, aquelas partes que não forem mais úteis devem ser descartadas para melhor aproveitamento da memória da máquina. A célula que possuir mais de um apontador apontando para ela só poderá ser liberada ou reusada quando não mais houver apontador algum a referenciando. Quando ela atingir esse estado, dizemos que é uma célula de lixo. Essas células são identificadas e recicladas por um programa específico chamado coletor de lixo[1]. O coletor de lixo para $\mathcal{LAMB}$ não está dentro do escopo de nosso trabalho.

[1] Do inglês, *garbage collector.*

Apêndice E

Utilização do Programa

Neste apêndice é apresentado como utilizar o compilador de $\mathcal{LAMB}$. Seu uso é bastante simples e direto, pois as duas fases de compilação, casamento de padrões e *lambda lifting* efetivo, ficam transparentes ao usuário de $\mathcal{SCRIPT}$, já que foi desenvolvido um arquivo em *shell script* [Thomas and Yates, 1988] para automatizar o processo.

A intenção do usuário ao executar o compilador é traduzir programas do cálculo lambda estendido em questão, $\mathcal{LAMB}$, para um cálculo de supercombinadores, $\mathcal{SUPER}$. A maneira mais direta de se fazer isso é, então, digitar uma linha de comando da forma

$$lamb2super\ exN.lamb$$

a fim de realizar a tradução.

No caso deste compilador, é exatamente assim que se dá. Contudo, alguns pontos devem ser ressaltados para que o usuário não seja traído pela simplicidade da utilização do programa:

1. o nome do programa, "lamb2super", sugere a tradução de um programa $\mathcal{LAMB}$ para um programa $\mathcal{SUPER}$. O programa "exN.lamb" é a saída do *Front-End* [Oliveira, 1998] de $\mathcal{SCRIPT}$, portanto é um arquivo binário de tokens. Já o arquivo gerado,"exemplo.super", representa a coleção de supercombinadores obtida a partir de "exN.lamb", é o fonte para o *Back-End* [Maia, 1994] de $\mathcal{SCRIPT}$. Sendo assim, a geração deste deve ser em arquivo texto, que é o tipo de arquivo lido pelo compilador de Maia.

2. Os prefixos dos arquivos, "ex", fazem alusão a EXemplo. A letra 'N' indica qual o número do exemplo que está sendo compilado. Esses dois caracteres, "ex", + 'N', são obrigatórios para que o programa processe sem erros e/ou resultados inesperados. São, portanto, no mínimo três caracteres obrigatórios para formar o nome do arquivo a ser traduzido.

3. Os arquivos a serem traduzidos devem estar no mesmo diretório do programa "bin2txt", que converte o arquivo binário gerado pelas Fases 1 e 2 em arquivo texto e de "fase1"

e "fase2", que são os programas para realizar casamento de padrões e *lambda lifting,* respectivamente. O motivo é, também, evitar erros.

4. A extensão ".lamb" não é obrigatória. Ela é colocada nos nomes dos arquivos apenas por razão de clareza.

5. Como o arquivo gerado será sempre "exemplo.super", é bom que o usuário se certifique da inexistência de um arquivo com mesmo nome no diretório de execução. Caso haja tal arquivo, ele deve ser excluído.

Exemplos:

- *lamb2super ex1.lamb:* CORRETO! O arquivo ex1.lamb é um arquivo de tokens e o arquivo gerado, ex1.super, é texto.
- *lamb2super ex1:* CORRETO! O arquivo ex1 apenas não apresenta a extensão.
- *lamb2super ex:* ERRADO! Foram utilizados apenas dois caracteres obrigatórios.
- *lamb2super ex30.lamb:* CORRETO!

Procedendo da maneira indicada, o usuário não terá problemas em executar esta etapa do compilador, pois os erros já foram eliminados pelo *Front-End* de $\mathcal{SCRIPT}$, e a operação de tradução é realizada sempre em programas corretos.

O Compilador para $\mathcal{SCRIPT}$

Conforme relatado no Capítulo 2, este trabalho encerra a construção do compilador para $\mathcal{SCRIPT}$. Para o usuário, então, é importante a constatação de que um programa $\mathcal{SCRIPT}$ irá se transformar em um código de Máquina-G, escrito em macros da linguagem C.

Com o intuito de cumprir esse objetivo, foi desenvolvido um "shell script" para integrar as três etapas, *Front-End, lambda-liting* e *Back-End,* do compilador. O comando é simples:

$$sc < arquivo\_de\_entrada >$$

onde devem haver alguns cuidados:

1. $< arquivo\_de\_entrada >$ é um arquivo $\mathcal{SCRIPT}$ texto;

2. os programas e arquivos do *Front-End* (Fase_Exec1, Fase_Exec2 e Fase_Exec3), do *lambda-lifting* (fase1, fase2, bin2txt e auxFuncs.txt) e do *Back-End* (comp_g) devem estar todos no mesmo diretório que o "script" *sc*.

Apêndice F

Outros Resultados Obtidos

Este apêndice mostra alguns exemplos de resultados obtidos com funções auxiliares geradas pelo *Front-End* do compilador de $\mathcal{SCRIPT}$. Para que haja uma leitura mais fácil dos resultados, o código das funções auxiliares gerado pelo *Front-End* é omitido em todos os exemplos, com exceção do primeiro. No resultado final, há também uma outra diferença: são acrescentados os supercombinadores auxiliares mencionados na Seção 4.1.1 cuja implementação pode ser vista no Apêndice G.

Código com funções auxiliares

```
Exemplo 1 -- LAMB

 LET TesteScript  =
  DEF -FUNCAO   =   LAM -FF   .   LAM -LISTA   .   LAM -FILTRO   .
   -LISTA   EQ   <   >   ->   <   >   ,   ( -LISTA   IS -XX   PRE -XXSS  -> (
   LET -XX    PRE -XXSS    = -LISTA
   IN   ( -FILTRO -XX   )   NE   TT   ->   <   >   CAT
    ( -FUNCAO -FF -XXSS -FILTRO   )   ,   ( -FF -XX   )   CAT
    ( -FUNCAO -FF -XXSS -FILTRO   )   )   ,   ?   )
  IN
  DEF -MAX-INDICE   =   LAM -LP   . -LP   IS   <   >   -> 0   ,
   -LP   IS   < -II   , -LL   >   PRE -LP1   ->
   (   LAM   < -II   , -LL   >   PRE -LP1   .
   LET -I-MAX    = -MAX-INDICE    ( -LP1    )
   IN -II   GR -I-MAX   -> -II   , -I-MAX   )   ( -LP   )   ,   ?
  IN
  DEF -COPY-LIST  =  LAM -LORIGEM  . LAM -LPARES  . LAM -II  . LAM -IMAX   .
   DEF -GET-PAIRS    =    LAM -LP1    .    LAM -KK    . -LP1  IS  <  >  ->
    < 0 , 0 > , -LP1    IS    < -KK1    , -VV1    >    PRE -LP2    ->
    (    LAM    < -KK1    , -VV1    >    PRE -LP2    . -KK    EQ -KK1    ->
    < -KK1  , -VV1  >  , -GET-PAIRS  ( -LP2  , -KK  )  )  ( -LP1  )  ,    ?
   IN -II   GR -IMAX   -> -LORIGEM   , -LORIGEM   IS   <   >   ->   <   >  ,
    -LORIGEM   IS -AA   PRE -LO1   ->   (   LAM -AA   PRE -LO1   .
   LET -VV    =
    LET     < -KK2     , -VV2     >     = -GET-PAIRS     ( -LPARES   , -II  )
    IN -KK2    EQ -II    -> -VV2    , -AA
   IN -VV   PRE -COPY-LIST   ( -LO1   , -LPARES   , -II   PLUS 1 , -IMAX ) )
   ( -LORIGEM   )   ,   ?
  IN
  LET ndom1   = 11
  ALSO q1   = "teste"
  ALSO t1   = "FF"
  IN
  LET t2   = ndom1   IS 12
  ALSO ndom2   = ndom1
  IN  < q1  , t1  >
```

```
Exemplo 1 -- LAMBASIC

 LET TesteScript  =
  DEF -FUNCAO   =   ( LAM nn4 . ( LAM -FF . ( LAM nn3 . ( LAM -LISTA .
   ( LAM nn2 . ( LAM -FILTRO   . -LISTA   EQ   <   >   ->   <   >  ,
   ($EMPTY -LISTA   ->   FF   ,   TT   AND   TT   ->   (
   LET nn1    = -LISTA
   IN
   LET -XX    =    ( $HEAD nn1    )
   IN
   LET -XXSS    =    ( $TAIL nn1    )
   IN   ( -FILTRO -XX   )   NE   TT   ->   <   >   CAT
    ( -FUNCAO -FF -XXSS -FILTRO   )   ,   ( -FF -XX   )   CAT
    ( -FUNCAO -FF -XXSS -FILTRO   )   )   ,   ?   )   )
    ( nn2   )   )   )   ( nn3   )   )   )   ( nn4   )   )
  IN
  DEF -MAX-INDICE   =   (  LAM nn8  .  (  LAM -LP  .  (  SIZE -LP  EQ 0  ) ->
   TT   ,   FF   -> 0   , $EMPTY -LP   ->   FF   ,  SIZE $HEAD -LP   NE 2  ->
   FF   ,   TT   AND   TT   AND   TT   ->
   (   (   LAM nn7   .   (   LAM -II   .   (   LAM -LL   .   (   LAM -LP1   .
   LET -I-MAX    = -MAX-INDICE    ( -LP1    )
   IN -II   GR -I-MAX   -> -II   , -I-MAX  ) ) ) ( ($HEAD nn7 ) EL 1 )
   ( ( $HEAD nn7 )  EL 2  )  ( $TAIL nn7  ) ) ) ( -LP ) , ? ) ( nn8 ) )
  IN
  DEF -COPY-LIST   =  ( LAM nn20 . ( LAM -LORIGEM . ( LAM nn19 . ( LAM -LPARES.
   (  LAM nn18  .  (  LAM -II  .  (  LAM nn17  .  (  LAM -IMAX  .
   DEF -GET-PAIRS    =    (    LAM nn12    .    (    LAM -LP1  . ( LAM nn11 .
    (    LAM -KK    .    (    SIZE -LP1    EQ 0    )    ->   TT  ,  FF  ->
    < 0    , 0    >    , $EMPTY -LP1  ->  FF  ,  SIZE $HEAD -LP1   NE 2   ->
    FF  ,  TT   AND    TT    AND   TT  ->  (  (  LAM nn10  .  (  LAM -KK1  .
    (  LAM -VV1  .  (  LAM -LP2  . -KK  EQ -KK1  ->  < -KK1    , -VV1    >  ,
    -GET-PAIRS  ( -LP2  , -KK  ) ) ) )  ( ( $HEAD nn10 ) EL 1  )
    ( ( $HEAD nn10  )  EL 2  )  ( $TAIL nn10 ) ) ) ( -LP1 )  ,  ? ) ( nn11 )))
    ( nn12 ) )
   IN -II   GR -IMAX -> -LORIGEM , ( SIZE -LORIGEM  EQ 0 ) ->  TT , FF  ->
    <  >   , $EMPTY -LORIGEM   ->   FF   ,   TT   AND   TT   -> ( ( LAM nn16 .
    (   LAM -AA   .   (   LAM -LO1   .
   LET -VV    =
    LET nn14     = -GET-PAIRS     ( -LPARES     , -II     )
```

```
  IN
  LET -KK2     =     ( nn14     EL 1     )
  IN
  LET -VV2     =     ( nn14     EL 2     )
  IN -KK2    EQ -II    -> -VV2    , -AA
 IN -VV   PRE -COPY-LIST   ( -LO1 , -LPARES , -II   PLUS 1 , -IMAX ) ) )
  ( $HEAD nn16 ) ( $TAIL nn16 ) ) ) ( -LORIGEM ) , ? ) ( nn17  ) ) )
  ( nn18   )   )   )   ( nn19   )   )   )   ( nn20   )   )
IN
LET ndom1   = 11
IN
LET q1   = "teste"
IN
LET t1   = "FF"
IN
LET t2   =   ( ndom1   NE 12   )   ->   FF   ,   TT
IN
LET ndom2   = ndom1
IN  < q1  , t1  >
```

```
Exemplo 1 -- SUPER

$S2-FF -LISTA -FILTRO = IF EQ -LISTA ( ) ( ) ( IF AP $EMPTY -LISTA FF
 IF AND TT TT (
 LET nn1  = -LISTA
 IN
 LET -XX  =  ( AP $HEAD nn1  )
 IN
 LET -XXSS  =  ( AP $TAIL nn1  )
 IN IF NE ( AP -FILTRO -XX ) TT CAT ( )
  ( AP AP AP $S1 -FF -XXSS -FILTRO ) CAT ( AP -FF -XX )
  ( AP AP AP $S1 -FF -XXSS -FILTRO ) ) ? )

$S3 -FF -LISTA nn2 = AP ( AP AP $S2 -FF -LISTA ) ( nn2 )

$S4 -FF -LISTA = ( AP AP $S3 -FF -LISTA )

$S5 -FF nn3 = AP ( AP $S4 -FF ) ( nn3 )

$S6 -FF = ( AP $S5 -FF )

$S7 nn4 = AP ( $S6 ) ( nn4 )

$S1 = ( $S7 )

$S9 -II -LP1 =
 LET -I-MAX  = AP $S8  ( -LP1  )
 IN IF GR -II -I-MAX -II -I-MAX

$S10 -II -LL = ( AP $S9 -II )

$S11 -II = ( AP $S10 -II )

$S12 nn7 = AP AP AP ( $S11 ) ( EL ( AP $HEAD nn7 ) 1 )
 ( EL ( AP $HEAD nn7 ) 2 ) ( AP $TAIL nn7 )

$S13 -LP = IF ( EQ SIZE -LP 0 ) TT IF FF 0 IF AP $EMPTY -LP FF
```

```
 IF NE SIZE AP $HEAD -LP 2 FF IF AND AND TT TT TT
 AP ( ( $S12 ) ) ( -LP ) ?

$S14 nn8 = AP ( $S13 ) ( nn8 )

$S8 = ( $S14 )

$S16 -GET-PAIRS -KK -KK1 -VV1 -LP2 = IF EQ -KK -KK1 ( -KK1 , -VV1 )
 AP -GET-PAIRS ( -LP2 , -KK )

$S17 -GET-PAIRS -KK -KK1 -VV1 =
 ( AP AP AP AP $S16 -GET-PAIRS -KK -KK1 -VV1 )

$S18 -GET-PAIRS -KK -KK1 = ( AP AP AP $S17 -GET-PAIRS -KK -KK1 )

$S19 -GET-PAIRS -KK nn10 = AP AP AP
 ( AP AP $S18 -GET-PAIRS -KK ) ( EL ( AP $HEAD nn10 ) 1 )
 ( EL ( AP $HEAD nn10 ) 2 ) ( AP $TAIL nn10 )

$S20 -GET-PAIRS -LP1 -KK = IF ( EQ SIZE -LP1 0 ) TT IF FF ( 0 , 0 )
 IF AP $EMPTY -LP1 FF IF NE SIZE AP $HEAD -LP1 2 FF
 IF AND AND TT TT TT AP ( ( AP AP $S19 -GET-PAIRS -KK ) ) ( -LP1 ) ?

$S21 -GET-PAIRS -LP1 nn11 = AP ( AP AP $S20 -GET-PAIRS -LP1 ) (nn11)

$S22 -GET-PAIRS -LP1 = ( AP AP $S21 -GET-PAIRS -LP1 )

$S23 -GET-PAIRS nn12 = AP ( AP $S22 -GET-PAIRS ) ( nn12 )

$S24 -LPARES -II -IMAX -GET-PAIRS -AA -LO1 =
 LET -VV  =
  LET nn14   = AP -GET-PAIRS   ( -LPARES   , -II   )
  IN
  LET -KK2   =   (   EL nn14 1   )
  IN
  LET -VV2   =   (   EL nn14 2   )
  IN IF  EQ -KK2 -II -VV2 -AA
 IN PRE -VV AP $S15 ( -LO1 , -LPARES , PLUS -II 1 , -IMAX )

$S25 -LPARES -II -IMAX -GET-PAIRS -AA =
```

```
 ( AP AP AP AP AP $S24 -LPARES -II -IMAX -GET-PAIRS -AA )

$S26 -LPARES -II -IMAX -GET-PAIRS nn16 = AP AP
 ( AP AP AP AP $S25 -LPARES -II -IMAX -GET-PAIRS )
 ( AP $HEAD nn16 ) ( AP $TAIL nn16 )

$S27 -LORIGEM -LPARES -II -IMAX =
 DEF -GET-PAIRS  =  ( AP $S23 -GET-PAIRS  )
 IN IF GR -II -IMAX -LORIGEM IF ( EQ SIZE -LORIGEM 0 ) TT IF FF ( )
  IF AP $EMPTY -LORIGEM FF IF AND TT TT AP
  (( AP AP AP AP $S26 -LPARES -II -IMAX -GET-PAIRS ))(-LORIGEM) ?

$S28 nn17 = AP ( AP AP AP $S27 -LORIGEM -LPARES -II ) ( nn17 )

$S29 -II = ( $S28 )

$S30 nn18 = AP ( $S29 ) ( nn18 )

$S31 -LPARES = ( $S30 )

$S32 nn19 = AP ( $S31 ) ( nn19 )

$S33 -LORIGEM = ( $S32 )

$S34 nn20 = AP ( $S33 ) ( nn20 )

$S15 = ( $S34 )

$S35 = 11

$S36 = "teste"

$S37 = "FF"

$S38 = IF ( NE $S35 12 ) FF TT

$S39 = $S35

$MOD_LAMB = ( $S36 , $S37 )
```

Código com funções auxiliares

```
Exemplo 2 -- LAMB

 LET TesteScript  =
  LET string   = "11"
  ALSO ndom1   = 11
  ALSO ndom2   = 11
  ALSO ldomN2   =
   LET -LISTA    =    LAM -NN    .
    DEF -LISTALOOP   =   LAM -MM  .  ( -MM GR -NN ) ->  <   > ,
     (-MM  PRE  ( -LISTALOOP  ( -MM   PLUS 1 ) ) )
    IN -LISTALOOP 10
   IN -LISTA 20
  ALSO ldomN3   =
   LET -LISTA    =    LAM -NN    .
    DEF -LISTALOOP   =   LAM -MM  .  ( -MM GR -NN ) ->  <   > ,
     ( -MM PRE ( -LISTALOOP   ( -MM  PLUS 1  ) ) )
    IN -LISTALOOP 11
   IN -LISTA 20
  IN
  LET ldomN1   =
   LET -LISTA    =    LAM -NN    .
    DEF -LISTALOOP   =   LAM -MM  .  ( -MM GR -NN ) ->  <   > ,
     ( -MM PRE ( -LISTALOOP   ( -MM  PLUS 1  ) ) )
    IN -LISTALOOP ndom1
   IN -LISTA ndom2
  IN  <  >

Exemplo 2 -- LAMBASIC


 LET TesteScript  =
  LET string   = "11"
  IN
  LET ndom1   = 11
  IN
  LET ndom2   = 11
  IN
```

```
  LET ldomN2   =
   LET -LISTA    =  ( LAM nn27 . ( LAM -NN .
    DEF -LISTALOOP  =  ( LAM nn25 . ( LAM -MM . ( -MM GR -NN ) -> <   > ,
     ( -MM PRE ( -LISTALOOP ( -MM  PLUS 1 ) ) ) ) ( nn25 ) )
    IN -LISTALOOP 10 )  ( nn27 ) )
   IN -LISTA 20
  IN
  LET ldomN3   =
   LET -LISTA    =  ( LAM nn32 . ( LAM -NN .
    DEF -LISTALOOP  =  ( LAM nn30 . ( LAM -MM . ( -MM GR -NN ) -> <   > ,
     ( -MM PRE ( -LISTALOOP ( -MM  PLUS 1 ) ) ) ) ( nn30 ) )
    IN -LISTALOOP 11 )  ( nn32 ) )
   IN -LISTA 20
  IN
  LET ldomN1   =
   LET -LISTA    =  ( LAM nn37 . ( LAM -NN .
    DEF -LISTALOOP  =  ( LAM nn35 . ( LAM -MM . ( -MM GR -NN ) -> <   > ,
     ( -MM PRE ( -LISTALOOP  ( -MM  PLUS 1 ) ) ) ) ( nn35 ) )
    IN -LISTALOOP ndom1 )  ( nn37 ) )
   IN -LISTA ndom2
  IN  <  >

Exemplo 2 -- SUPER


$S35 = "11"

$S36 = 11

$S37 = 11

$S40 -NN -LISTALOOP -MM = IF ( GR -MM -NN ) ( ) ( PRE -MM
 ( AP -LISTALOOP ( PLUS -MM 1 ) ) )

$S41 -NN -LISTALOOP nn25 = AP ( AP AP $S40 -NN -LISTALOOP ) ( nn25 )

$S42 -NN =
 DEF -LISTALOOP  =  ( AP AP $S41 -NN -LISTALOOP  )
 IN AP -LISTALOOP 10
```

```
$S43 nn27 = AP ( $S42 ) ( nn27 )

$S39 = ( $S43 )

$S38 = AP $S39 20

$S45 -NN -LISTALOOP -MM = IF ( GR -MM -NN ) ( )
 ( PRE -MM ( AP -LISTALOOP ( PLUS -MM 1 ) ) )

$S46 -NN -LISTALOOP nn30 = AP ( AP AP $S45 -NN -LISTALOOP ) ( nn30 )

$S47 -NN =
 DEF -LISTALOOP  =  ( AP AP $S46 -NN -LISTALOOP  )
 IN AP -LISTALOOP 11

$S48 nn32 = AP ( $S47 ) ( nn32 )

$S44 = ( $S48 )

$S44 = AP $S39 20

$S50 -NN -LISTALOOP -MM = IF ( GR -MM -NN ) ( )
 ( PRE -MM ( AP -LISTALOOP ( PLUS -MM 1 ) ) )

$S51 -NN -LISTALOOP nn35 = AP ( AP AP $S50 -NN -LISTALOOP ) ( nn35 )

$S52 -NN =
 DEF -LISTALOOP  =  ( AP AP $S51 -NN -LISTALOOP  )
 IN AP -LISTALOOP $S36

$S53 nn37 = AP ( $S52 ) ( nn37 )

$S49 = ( $S53 )

$S49 = AP $S39 $S37

$MOD_LAMB = ( )
```

Código com operador de composição de funções ';'

```
Exemplo 3

-- LAMB

 LET TesteScript  =
  LET f   =   LAM a   . a   PLUS 1
  ALSO g   =   LAM b   . b   PLUS 1
  IN
  LET c   = f   ; g   ; 3
  IN  <  >


-- LAMBASIC

 LET TesteScript  =
  LET f   =   (   LAM nn22   .   (   LAM a   . a   PLUS 1   )   ( nn22   ) )
  IN
  LET g   =   (   LAM nn24   .   (   LAM b   . b   PLUS 1   )   ( nn24   ) )
  IN
  LET c   = f   ( g   ( 3   )   )
  IN  <  >

-- SUPER


$S36 a = PLUS a 1

$S37 nn22 = AP ( $S36 ) ( nn22 )

$S35 = ( $S37 )

$S39 b = PLUS b 1

$S40 nn24 = AP ( $S39 ) ( nn24 )

$S38 = ( $S40 )
```

```
$S41 = AP $S35 ( AP $S38 ( 3 ) )

$MOD_LAMB = ( )
```

Apêndice G

Supercombinadores Auxiliares

```
$EMPTY x = IF EQ SIZE x 0 TT FF

$PREFIX n v = IF EQ n 0 ( ) PRE AP $HEAD v
               AP $PREFIX (MINUS n 1) AP $TAIL v

$N2Q n = IF EQ (DIV n 10) 0 AP $QUOTE n
         ( AUG (AP $N2Q DIV n 10)  AP $QUOTE (REM n 10))

$QUOTE d = IF EQ d 0 '0' IF EQ d 1 '1' IF EQ d 2 '2' IF EQ d 3 '3'
            IF EQ d 4 '4' IF EQ d 5 '5' IF EQ d 6 '6' IF EQ d 7 '7'
             IF EQ d 8 '8' IF EQ d 9 '9' '?'

$T2Q t = IF EQ t TT ('T','T') IF EQ t FF ('F','F') ?

$Q2Q q = IF (EQ SIZE q 0) ( ) (CAT (AP $HEAD q) (AP $Q2Q AP $TAIL q))

$VAL x = x

$HEAD x = IF GR SIZE x 0 EL x 1 ?

$TAIL x n = IF LS SIZE x n () PRE EL x n AP $TAIL x n+1

$TRUTH q =  IF NE SIZE q 2 ? IF AND EQ EL q 1 'T' EQ EL q 2 'T' TT
             IF AND EQ EL q 1 'F' EQ EL q 2 'F' FF ?
```