Carlos Henrique Cabral Duarte

O desenvolvimento de um compilador MIRANDA usando um método orientado a objetos

Dissertação apresentada ao Departamento de Informática da PUC/RJ como parte dos requisitos para obtenção do título de Mestre em Ciência da Computação

Orientador: *Prof. Roberto Ierusalimschy*

DEPARTAMENTO DE INFORMÁTICA
PONTIFíCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

29 de julho de 1994

Agradecimentos:

- Ao professor Roberto Ierusalimschy, orientador deste trabalho, que teve a paciência de conduzir meus estudos sem deixar que estes se tornassem algo etéreo e sem objetividade, o que às vezes faço sem perceber.
- Ao professor Carlos Lucena, pelas excelentes idéias ao longo do curso e pelo incentivo constante para que continuasse estudando.
- Aos demais professores com os quais tive contato direto ao longo do mestrado, que muito ajudaram no meu amadurecimento na área que escolhi estudar: Hermann, Júlio César, Rangel e Rubens.
- A meus professores de graduação, que souberam me dar ateriormente a formação adequada com tão poucos recursos e experiência.
- Ao CNPq e à P.U.C., que propiciaram os recursos financeiros e materiais necessários para que eu pudesse realizar o curso.
- Aos funcionários da secretaria, laboratório e biblioteca, e em especial ao Luiz Cláudio, que sempre conseguiu os textos necessários quando precisei deles.
- À Patrícia e ao Matheus, pela compreensão do quão importante este curso era para mim e pelo incentivo nos momentos em que meu ânimo para continuar diminuia.
- Aos meus pais Dione e Aracy, minha irmã Gisele e sobrinha Thais, pela preocupação e apoio que sempre demonstraram.
- Aos amigos do B.N.D.E.S., pelo incentivo para que concluisse esta dissertação.

Resumo

O desenvolvimento de compiladores para linguagens funcionais de programação é uma atividade complexa, que demanda um desenho cuidadoso, objetivando evitar que o produto final seja ineficiente, e, por outro lado, permitindo que este possa ser mantido e extendido de maneira simples. Neste trabalho descrevemos um compilador para a linguagem funcional Miranda, desenvolvido através de um método que conjuga orientação a objetos como enfoque para o processo de desenvolvimento à especificação formal de linguagens de programação como técnica de desenho, tornando mais tratável e sistematizada tal atividade.

Palavras chave: compiladores, linguagens funcionais, orientação a objetos, desenvolvimento de software

Abstract

The development of functional languages compilers is a complex activity, which demands a careful design to avoid an ineficient final product and to allow it to be mantained and extended in a simple manner. In this paper we describe the development of a compiler for Miranda functional language, developed using a method which joins object orientation as a development process approach and formal specification of programming languages as a design technique, changing this activity into a more tractable and systematic one.

Keywords: compilers, functional languages, object orientation, software development

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

> *"The core of this case-study will challenge mathematical formalism, but will not challenge directly the ultimate positions of matematical dogmatism. Its modest aim is to elaborate the point that informal, quasi-empirical, mathematics does not grow through a monotonous incrase of the number of undubitably established theorems, but through the incessant improvement of guesses by speculation and criticism, by the logic of proofs and refutation"*
>
> *Irme Lakatos*

A construção de compiladores é uma atividade das mais antigas em Ciência da Computação, apresentando problemas já bastante explorados, como as análises léxica e sintática, e a geração de código. Estes podem ter uma implementação realizada facilmente pela aplicação de métodos, técnicas e ferramentas amplamente difundidos.

Por outro lado, a representação de programas e seu o processo de tradução por compiladores geralmente são menos abordados, por possuírem muitas características dependentes tanto das linguagens de programação quanto dos formalismos usados para definir sua semântica. O desenvolvimento de componentes de software para estes fins não é realizado de maneira sistemática, rigorosa ou organizada em muitos casos. Este processo de desenvolvimento foi escolhido como objeto de estudo desta dissertação.

Este trabalho aborda a construção de compiladores sob a ótica de Engenharia de

Software. Pretende-se atacar este problema usando uma perspectiva formal, sem entretanto tratar de aspectos relacionados à corretude do software, que envolvem a elaboração de teoremas e provas custosas, e que, na maioria dos casos abordados aqui, já foram desenvolvidos por outros autores.

Escolhemos realizar o desenvolvimento de um compilador para uma linguagem funcional, usando as diretrizes apresentadas acima. O que desejamos é que esta atividade possa ser extensa e real o suficiente para que um conjunto de regras possa ser definido, e usado posteriormente para construir melhores compiladores, especificados por um conjunto de contruções formais que permitam a comprovação ou refutação de suas propriedades.

1.1 Especificação Formal de Linguagens de Programação e Orientação a Objetos

Na construção de compiladores, entre as ferramentas mais utilizadas está a especificação formal de linguagens de programação. Especificações deste tipo servem como referências para usuários e implementadores, tornando possível também a implementação automática de compiladores e a prova de propriedades destes softwares e dos programas manipulados por eles, como mostrado por Pagan em [Pag81]. Além disso, levam à padronização sintática e semântica das linguagens.

Infelizmente, tratamentos formais não são satisfatórios em muitos casos, devido a complexidade inerente a esta atividade, a inadequações notacionais e a especificações incorretas, como notado por Goguen em [GM87]. Estes fatos aumentam a complexidade das construções criadas ao longo do processo de desenvolvimento, tornando-o mais custoso e demorado.

Um enfoque que vem sendo usado para reduzir complexidade no processo de desenvolvimento de software é orientação a objetos [WBJ90], permitindo que o trabalho seja melhor estruturado e até reutilizado, através de conceitos como herança, composição e polimorfismo.

Entretanto, como é um enfoque informal, qualquer construção que utilize os conceitos provenientes da orientação a objetos está sujeita a má interpretação. Isto ocorre também porque os próprios conceitos utilizados são interpretados de maneiras diferentes por diversos autores.

Tal enfoque e tal técnica podem ser perfeitamente combinados, completando as deficiências de um com as virtudes de outro.

1.2 Objetivos e tese do trabalho

A principal tese defendida nesta dissertação é que orientação a objetos, usada como enfoque para o processo de desenvolvimento de software, conjugada à especificação formal de linguagens de programação, usada como técnica de desenho, permite que a construção de compiladores seja feita de forma organizada e sistematizada.

Pretendemos mostrar que dessa forma produtos podem ser criados com características bastante desejáveis. Estes devem ser:

1. **organizados**: toda a clareza conceitual do desenho deve ser preservada na implementação do compilador;
2. **modulares**: a funcionalidade dos componentes e a interface entre eles deve estar bem definida, fazendo que alterações em cada um tenham efeitos localizados e controlados;
3. **extensíveis**: o aumento da funcionalidade pode ser feito sem ferir as duas características anteriores.

Para que isto possa ser mostrado, escolhemos um estudo de caso com tamanho mediano mas com complexidade elevada, que apresenta inclusive alguns desafios não exclarecidos completamente pela literatura da área: um compilador para a linguagem funcional Miranda[1]. Nosso objetivo então é aproveitar as oportunidades oferecidas para a aplicação da idéia acima no desenvolvimento deste compilador, procurando obtender ganhos em relação a outras propostas.

[1]Miranda é uma marca registrada da Research Software Limited

Usaremos as especificações propostas por Peyton-Jones em [Jon87] para desenhar e implementar tal compilador, corrigindo, completando ou alterando-as quando necessário, procurando alcancar uma implementação conceitualmente clara, modular e extensível.

1.3 Organização do texto

O restante deste trabalho está organizado em três capítulos e um apêndice. A longo do texto, as seguintes convenções notacionais serão usadas:

- **texto em negrito**: representa instâncias de classes (objetos).
- `texto type-writer`: representa programas em uma das linguagens utilizadas.
- *texto itálico*: representa especificações.

No segundo capítulo será apresentado um método para desenho de compiladores. Serão descritas regras para derivar, a partir da especificação formal da linguagem de programação, a estrutura de classes usada para representar os programas no compilador. Além disso, um *framework* usado no desenvolvimento dos componentes independentes das linguagens usadas será descrito.

Um compilador para a linguagem Miranda, desenvolvido usando tal método, será então apresentado no terceiro capítulo. As várias linguagens usadas na compilação serão descritas em conjunto com os esquemas de tradução que definem os mapeamentos entre elas. Usando estes esquemas, técnicas convencionais para compilação de linguagens funcionais foram especificadas, como o casamento de padrões, a análise de dependências e o *lambda lifting*, e estas serão demonstradas através de exemplos. Para a linguagem de mais baixo nível será apresentado também um interpretador e um depurador.

Finalmente, no capítulo quatro serão apresentadas as nossas conclusões, as relações deste com outros trabalhos e as extensões futuras que podem ser realizadas. No apêndice são exibidas as sintaxes das linguagens usadas ao longo do processo de compilação.

Capítulo 2

Um método para desenho de compiladores

"Formalization is an experimental science".

Dana Scott

Qualquer atividade de formalização está fortemente fundamentada na experimentação. Em outras ciências é freqüente uma teoria começar a ser usada e através desta prática evidenciar sua validade, levando à criação de uma definição rigorosa (e em alguns casos formal) para tal, por especialistas.

Em Ciência da Computação este fato é amplamente aceito, pois novos enfoques, técnicas e tecnologias aparecem de forma muito rápida, sendo necessário transpô-los para os métodos já existentes ou criar novos métodos que as incorporem. Em linhas gerais, o presente capítulo versa sobre este assunto: a definição de um novo método a partir da experimentação.

Nosso objetivo é delinear, apresentando de forma rigorosa os vários experimentos que foram realizados e sistematizando os resultados, um método para construção de compiladores que utiliza tanto orientação a objetos como enfoque para o processo de desenvolvimento quanto especificação formal de linguagens de programação como técnica de desenho. Mais precisamente, desejamos mostrar como um compilador pode ser projetado e implementado de maneira eficaz usando orientação a objetos, dada uma especificação formal da linguagem.

As especificações formais utilizadas devem ser compostas pela declaração da gramática da respectiva linguagem, na forma de *Bacus-Naur Form* (BNF), com todas suas ambiguidades já resolvidas, e por um conjunto de esquemas ou regras de tradução, que descrevam o significado de cada símbolo da linguagem em função de símbolos com significado já definido. A notação utilizada para representar tais esquemas será semelhante a apresentada por Peyton-Jones em [Jon87].

A partir das especificações, mostraremos como derivar estruturas de classes e métodos para cada uma destas classes, que implementam as estruturas de dados e a funcionalidade do compilador, respectivamente. Para facilitar a compreensão sobre como são estruturadas estas classes, será usada a notação gráfica proposta por Rhumbauh em [R$^+$91], escolhida devido a sua simplicidade.

2.1 Sintaxe

A especificação da gramática de uma linguagem é usada normalmente na construção de um analisador léxico e sintático, que, ao ser executado, irá tomar como entrada um programa escrito na linguagem fonte e irá gerar uma representação interna fiel a este programa, a ser usada nos passos posteriores de compilação.

Geralmente, a implementação de analisadores de programas é feita por automatos guiados por tabelas, que fazem a leitura do texto e o traduzem em uma representação interna que depende da sintaxe da linguagem de programação em questão. A criação de componentes que realizem as primeiras análises pode ser resolvida satisfatoriamente usando geradores de analisadores léxicos e sintáticos (por exemplo LEX e YACC).

Resta-nos mostrar como criar estruturas de dados que representem os programas. Isto pode ser feito com base na especificação da gramática da linguagem. Como esta descreve a estrutura de todos os programas aceitos como sintaticamente corretos pelo analisador, cada símbolo de cada um destes programas deverá ser representado na estrutura de dados escolhida. Cada regra de produção deverá indicar como conjuntos destas estruturas devem ser organizados.

Entretanto, usando orientação a objetos como enfoque, existem várias formas de estruturar os dados. Crenshaw em [Cre91] identificou algumas regras para representar símbolos de programas dessa forma: "Para identificar os objetos requeridos por um tradutor, olhe para as equações sintáticas. Essencialmente, todo não-terminal representa um objeto."

Considere como exemplo a seguinte produção, que descreve como listas são definidas em Miranda (a definição completa da sintaxe da linguagem é apresentada no apêndice):

$$
\begin{array}{lll}
list & \rightarrow & string \\
list & \rightarrow & head\_tail\_list \\
list & \rightarrow & empty\_list \\
list & \rightarrow & enumerated\_list \\
list & \rightarrow & list\_comprehension
\end{array}
$$

Usando a regra apresentada acima, *list*, *string*, *head_tail_list*, *empty_list*, *enumerated_list* e *list_comprehension* seriam representados em tempo de execução por objetos pertencentes às classes *List*, *String*, *HeadTailList*, *EmptyList* e *ListComprehension*, respectivamente.

Em seu trabalho, Crenshaw não considera a existência de uma especificação da semântica da linguagem. Assim, nada impede que possam existir símbolos terminais que tenham que aparecer na representação interna. Isto ocorreria com STRING e EMPTY_LIST se escolhessemos representar listas em Miranda como no exemplo abaixo [1]:

$$
\begin{array}{lll}
list & \rightarrow & \mathrm{STRING} \\
list & \rightarrow & expression\ \mathrm{CONS}\ expression \\
list & \rightarrow & \mathrm{EMPTY\_LIST} \\
list & \rightarrow & \mathrm{LB}\ contents\_list\ \mathrm{RB} \\
list & \rightarrow & \mathrm{LB}\ contents\_list\ \mathrm{VBAR}\ qualifier\_list\ \mathrm{RB}
\end{array}
$$

[1]Os símbolos terminais são exibidos em maiúsculas.

Neste trabalho, entretanto, existem esquemas de tradução que dão significado à sintaxe abstrata da linguagem, ou seja, àqueles símbolos que têm semântica, a ser observada na execução dos respectivos programas. Portanto é razoável exigir (para organizacionar melhor o compilador) que apenas os símbolos desta sintaxe abstrata sejam representados como não-terminais.

Existem ainda outras informações na especificação da gramática. Como usá-las? Estas informações se referem à construção dos símbolos, e portanto podem nos dizer como estão relacionadas as estruturas de dados que os representam.

A definição de várias regras de produção para o mesmo símbolo indica que aqueles presentes do lado direito de uma destas são categorias sintáticas que podem ser usadas para substituir o termo que aparece do seu lado esquerdo em um contexto mais genérico. Esta característica pode ser facilmente modelada através do conceito de herança. Assim, este caso deve levar a criação de uma classe, representando o símbolo do lado esquerdo, da qual cada classe que representa um símbolo que aparece do lado direito das produções é sub-classe. No exemplo acima, *String*, *HeadTailList*, *EmptyList*, *EnumeratedList* e *ListComprehension* serão sub-classes de *List*.

É de se esperar que produções onde símbolos são definidos por agregação dêem origem a uma classe, representando o símbolo do lado esquerdo, composta pelas classes que representam os símbolos que aparecem do lado direito desta produção. A partir do exemplo abaixo

$$head\_tail\_list \quad \rightarrow \quad expression \text{ CONS } expression$$

deve ser devivada a classe *HeadTailList*, composta por duas ocorrências de *Expression*, representando os respectivos símbolos agregados que aparecem na produção.

Em alguns casos, o compartilhamento de estruturas conceituais entre as classes pode ser aumentado, generalizando os componentes comuns. Seguindo os passos apresentados anteriormente, das produções a seguir

$$enumerated\_list \rightarrow \text{LB } contents\_list \text{ RB}$$
$$list\_comprehension \rightarrow \text{LB } contents\_list \text{ VBAR } qualifier\_list \text{ RB}$$

seriam derivadas as classes *EnumeratedList* e *ListComprehension*, sub-classes de *List* (devido ao exemplo anterior), que possuem *ContentsList* em comum. Em casos análogos a este uma nova classe deve ser gerada, chamada *DefinedList* para o exemplo anterior, possuindo como componente apenas *ContentsList*.

Generalizando ainda mais os conceitos comuns que aparecem em *BNFs*, podemos dar um tratamento diferenciado a definições que são listas de símbolos. A representação destas ficaria melhor encapsulada em uma classe especial, chamada *ProgramSymbolList* no nosso caso, da qual toda classe que representa uma lista de símbolos seria sub-classe. Do exemplo abaixo seria derivada a classe *ContentsList*, sub-classe de *ProgramSymbolList*, e com componente *ListContents*.

$$contents\_list \rightarrow list\_contents$$
$$contents\_list \rightarrow list\_contents \text{ COMMA } contents\_list$$

Sumarizando o que foi apresentado nesta seção, temos:

na Gramática	**na Estrutura**
símbolo não-terminal	classe abstrata
símbolo terminal	desprezado
produção	classe concreta (subclasse da criada para representar o lado esquerdo e composição daquelas do lado direito)

Finalmente, a estrutura de classes relacionada ao símbolo *list* é apresentada na figura 2.1. Nesta, triângulos e losangos representam herança e composição, respectivamente.

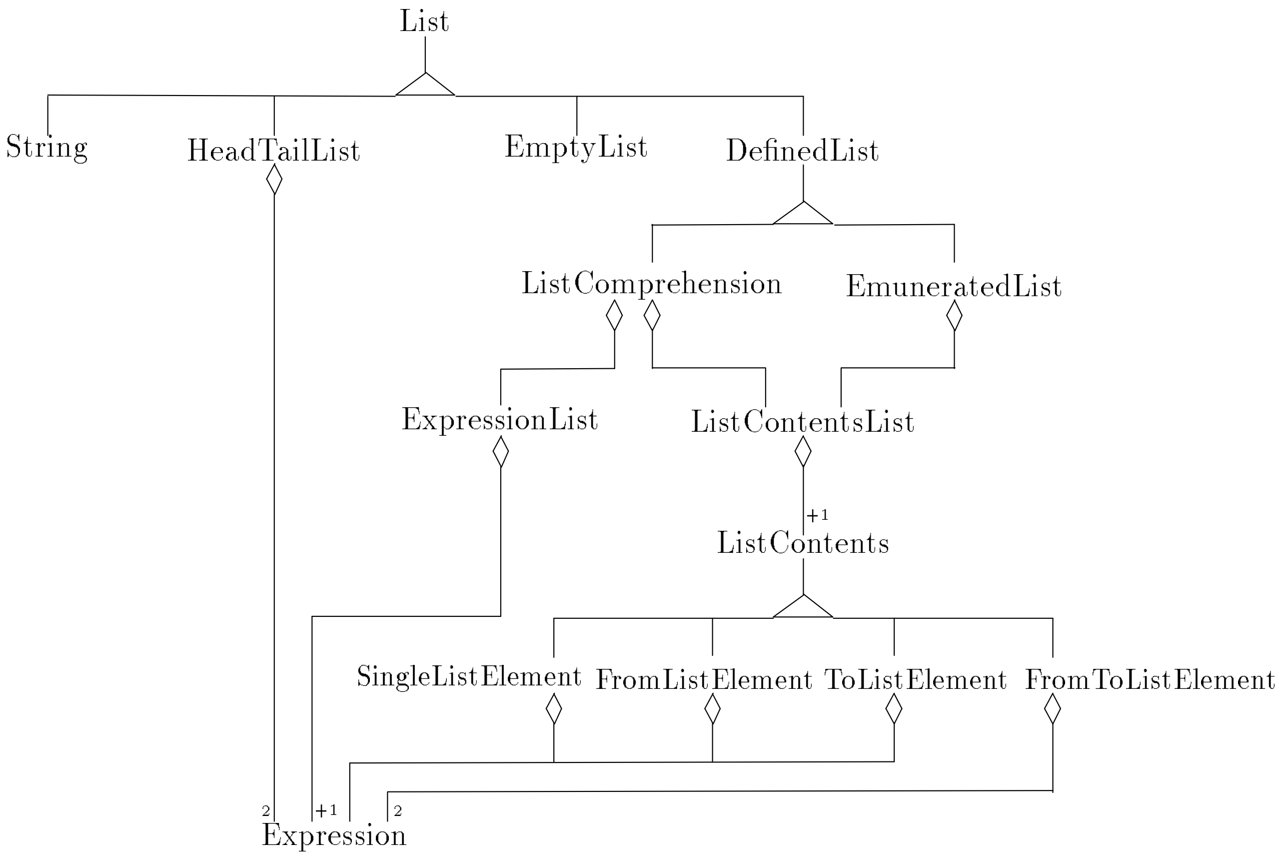

Figura 2.1: **Estrutura das classes relacionadas a List**

2.2 Semântica

Uma vez criada uma estrutura de dados que represente os símbolos da gramática, é necessário pensar em como representar a semântica da linguagem. Novamente, Crenshaw propõe uma regra para tal [Cre91]: "Toda entrada na tabela de símbolos é um objeto. Este tem um nome e uma ação a ser realizada. Um vez identificado pelo nome, ele precisa apenas de uma mensagem: monte a si mesmo."

Observando tal regra, aparecem algumas questões que devem ser resolvidas: somente os objetos que aparecem na tabela de símbolos têm semântica? Qual a estrutura das mensagens às quais estes objetos devem dar resposta? Como é realizada esta resposta? Como estas mensagens estão relacionadas entre si? No nosso caso, existe uma especificação formal da semântica da linguagem, e esta pode ser usada para responder tais questões.

O que pretendemos com a definição de esquemas formais de tradução é especificar a semântica da linguagem exibindo funções que mapeiem cada símbolo em um outro, com significado já conhecido. Assim, cada esquema estabelece uma funcão, que tem normalmente como domínio símbolos da gramática da linguagem fonte e como imagem símbolos da gramática da linguagem objeto (não necessariamente disjuntas). Cada esquema é definido por uma assinatura e um conjunto de regras, onde do lado esquerdo aparecem termos do domínio e do lado direito termos da imagem, representados usando as respectivas sintaxes.

Temos em mãos hierarquias de classe que representam domínios de esquemas de tradução, como mostrado na seção anterior. É necessário então definir hierarquias de classe que representem as imagens desses esquemas. Considerando que o programa apresentado ao compilador será traduzido sucessivamente em programas equivalentes a ele, escritos em outras linguagens, até que uma representação executavel pela máquina seja alcançada, pode-se usar as mesmas regras de representação apresentadas na seção anterior para a linguagem objeto.

Dessa forma, esquemas de tradução podem ser representados naturalmente como métodos (as mensagens a que Crenshaw se referia) em cada uma das classes. Esta representação é extremamente fiel, já que uma das características principais desses esquemas é o seu polimorfismo, e o mesmo ocorre com métodos de classes organizadas sob a forma de herança.

Exemplificando, considere uma lista definida pelos seus componentes, como abaixo:

$$\mathit{defined\_list} \quad \rightarrow \quad \mathit{enumerated\_list} \ \mid \mathit{list\_comprehension}$$

$$\begin{aligned}
&\mathbf{TE} : \mathit{defined_list} \rightarrow \mathit{lambda_expr}\\
&\mathbf{TE}[\mathit{enumerated_list(contents_list)}] \triangleq \mathbf{TE}[\mathit{contents_list}]\\
&\mathbf{TE}[\mathit{list_comprehension}] \triangleq \mathbf{TQ}[\mathit{list_comprehension}](\mathrm{NIL})
\end{aligned}$$

A semântica deste símbolo é definida por **TE**. Quando este é uma enumeração, tem como significado a tradução do seu componente *contents_list*. Caso este seja uma lista definida pelas propriedades de seus elementos, então terá significado dado pelo esquema **TQ**. Neste exemplo, a representação de **TE** seria o método `TE`, nas classes *DefinedList*, *EnumeratedList* e *ListComprehension*, que teria como resultado ao ser computado um objeto da classe *LambdaExpr*.

Ainda no exemplo acima, podemos ver que **TE** utiliza apenas informações oriundas da sua aplicação sobre os componentes do elemento no qual foi aplicada. Isto pode ser visto na assinatura do esquema ($\rightarrow$ *lambda_expr*) se o elemento da liguagem fonte não for considerado). Porém, existem casos em que esquemas de tradução necessitam conhecer outros elementos da linguagem objeto, resultantes de aplicações anteriores de esquemas de tradução. Isto é o que ocorre com **TQ**.

Na figura 2.2 mostramos a definição de **TQ**, que atua sobre um elemento *list_comprehension* e sobre um elemento *lambda_expr*, resultante de uma aplicação anterior da mesma regra. A representação deste tipo de esquema de tradução exige a definição de um método na respectiva classe que receba parâmetros: `TQ` seria um método de *ListComprehension* recebendo objetos *LambdaExpr*.

As representações para o significado de termos de uma linguagem se torna bastante intuitiva dadas as regras acima. A implementação dos esquemas de tradução pode ser projetada e realizada sem a necessidade de conceitos adicionais. Para isto, basta definir os respectivos métodos tomando a representação interna correspondente ao lado esquerdo de todas as regras de cada esquema.

Contudo, uma especificação formal não se compõe apenas de esquemas. Em geral é necessária a definição de funções auxiliares, como `Match` no exemplo anterior. Este tipo de função associa entre si elementos da mesma linguagem. Elas podem ser representadas dissociadas de qualquer hierarquia de classes, tendo como domínio e imagem símbolos da mesma linguagem.

$$\begin{aligned}&\textbf{Match} : \textit{lambda\_expr\_list} \times \textit{lambda\_expr\_list} \times \\ &\qquad \textit{lambda\_expr\_list} \times \textit{lambda\_expr} \rightarrow \textit{lambda\_expr} \\ &\textbf{Match}[.](...) \triangleq \text{/* } \textit{veja no proximo capitulo} \text{ */}\end{aligned}$$

$list\_comprehension \quad \rightarrow \quad$ LB *contents_list* VBAR *qualifier_list* RB

TQ : $list\_comprehension \times lambda\_expr \rightarrow lambda\_expr$

TQ[LB *contents_list* VBAR *pattern* IN *list expression_list* RB]*(lambda_expr)* $\triangleq$
letrec h = LAMBDA *us* DOT
*(*CASE *(((*IF *((*EQUAL *us)* NIL*)) lambda_expr)* FAIL*)*
*(***let** y = *(*SELECT-2-2 *us), z* = *(*SELECT-2-1 *us)* **in**
Match[.]*((x,),* **TE**[*pattern*],
*((***TQ**[*list_comprehension*]*(h y) z),), (h y))))*
in *(h* **TE**[*list*]*)*

TQ[LB *contents_list* VBAR *expression expression_list* RB]*(lambda_expr)* $\triangleq$
IF **TE**[*expression*]
TQ[LB *contents_list* VBAR *expression_list* RB]*(lambda_expr)*
lambda_expr

TQ[LB *contents_list* VBAR RB]*(lambda_expr)* $\triangleq$
*(***TE**[*contents_list*] *lambda_expr)*

Figura 2.2: **Esquema de compilação TQ**

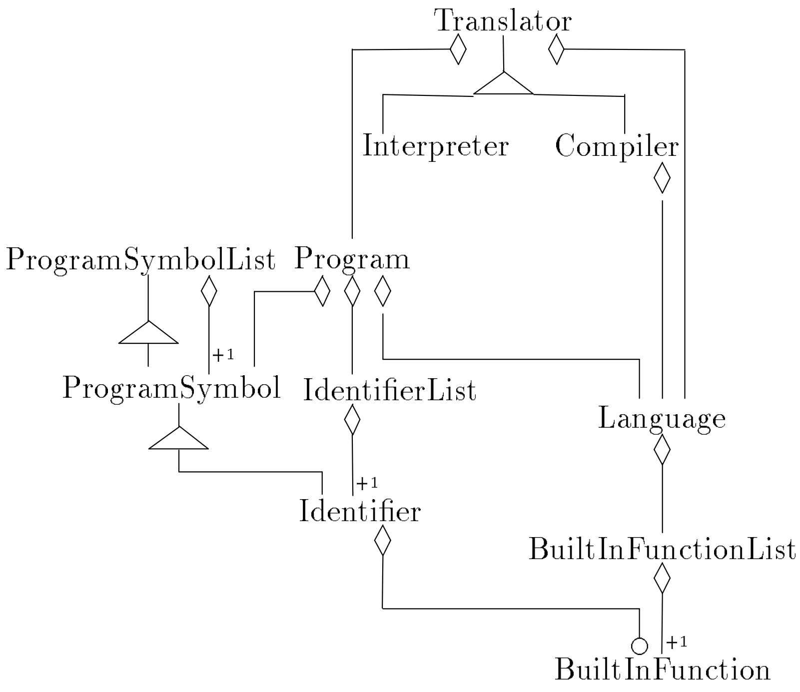

Figura 2.3: **Estrutura de classes para desenvolvimento de compiladores**

2.3 Um *Framework* para apoio ao desenvolvimento de compiladores

Como mostramos nas seções anteriores, a especificação formal de uma linguagem de programação pode ser usada como base para o desenho dos componentes de um compilador que são dependentes desta linguagem. Porém, existem vários outros componentes que são independentes da linguagem, e portanto podem receber um tratamento mais genérico.

Ao ser usada orientação a objetos como enfoque, o tratamento de componentes ou sub-sistemas genéricos visando a reutilização de desenho ou de implementações é feito através da criação de conjuntos de classes. Estes conjuntos de classes são denominados estruturas (*frameworks*), e seu uso se dá principalmente através do refinamento de seus elementos (herança), ou através do uso de seus elementos na definição de novas classes (composição) [WBJ90].

Uma estrutura de classes foi definida e implementada, visando auxiliar o desenho e implementação de componentes de compiladores que independem da linguagem de programação. Nosso método prescreve então a (re)utilização dessas classes na criação de compiladores específicos. Parte deste *framework* é apresentado na figura 2.3.

Na figura estão representados basicamente os elementos da estrutura de classes responsáveis pelos aspectos organizacionais de um tradutor. Tradutores podem ser compiladores ou interpretadores, que se utilizam de linguagens e programas para operar. A diferença principal entre as duas classes de tradutores é que os primeiros operam com objetivo de determinar qual a representação de um programa na linguagem objeto, dado um programa na linguagem fonte, enquanto que os segundos operam com objetivo de apresentar um comportamento observável para o programa apresentado como entrada escrito na linguagem fonte. Ambas classes possuem um método que cuida da re-escrita do programa original.

Programas possuem entre seus componentes uma representação na forma de árvore, composta por objetos que são instâncias das classes que representam os símbolos da linguagem em questão, derivadas usando este método. Estas devem ser todas sub-classes de *ProgramSymbol* ou *ProgramSymbolList*. Esta representação foi escolhida para encapsular um tratamento genérico para a estrutura e a funcionalidade comuns a estes objetos, que são a leitura, gravação, identificação e coleta de lixo.

Os elementos que são folhas na árvore que representa um programa devem ser os terminais da gramática, e entre eles estão os identificadores. Estes são símbolos de programa especiais, que representam variáveis, funções e tipos, entre outros, e tem uma organização (tabela de símbolos) própria e independente, usada nos casos necessários. Um desses casos é a possibilidade, especificada na figura como uma circunferência acima de `BuiltinFunction`, de identificadores representarem funções pré-definidas, o que pode ser visto pela composição de `Identifier`.

Conjuntos de funções pré-definidas (`BuiltinFunctionList`) são atributos de cada linguagem criada usando este *framework*. Cada uma destas funções representa um trecho de código, escrito em uma linguagem de nível mais baixo. Durante o processo de tradução de programas pelos respectivos compiladores, ao ser encontrado um identificador que representa uma destas funções, é acrescentado ao programa traduzido o trecho de código equivalente.

Instâncias da classe *ProgramSymbol* podem ser lidas ou gravadas em disco, ficando

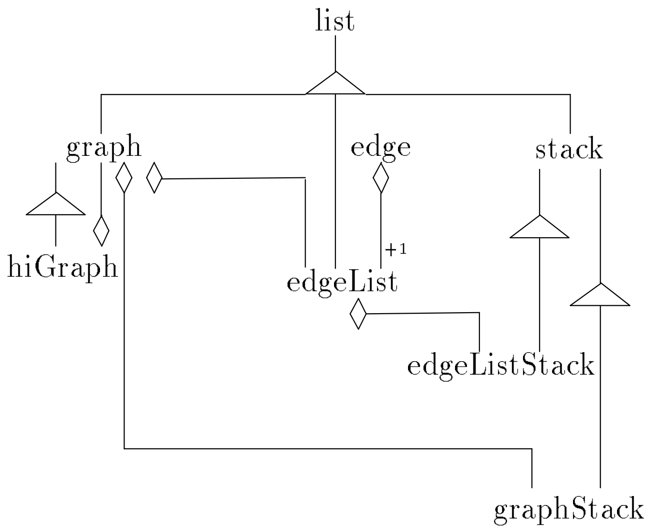

Figura 2.4: **Estrutura de classes genéricas**

com a responsabilidade de ler ou gravar os objetos que as compõem, ou por decidir qual instância de classe deve sofrer esta ação, quando ocorre herança entre elas. Funções pré-definidas são armazenadas e recuperadas dessa forma quando necessário, e o mesmo ocorre com a representação de cada programa. Para realizar tais atividades, basta que o método de leitura ou gravação da classe que representa o símbolo requerido seja ativado, carregando assim o trecho de código correspondente.

Além das classes apresentadas na figura 2.3, é necessário também definir classes básicas para qualquer implementação orientada a objetos, que encapsulam o comportamento de estruturas de dados como listas, pilhas, grafos e outras. Estas estão representadas na figura 2.4.

2.4 Implementações usando este método

A aplicação do método apresentado neste capítulo permite organizar satisfatoriamente um analisador léxico e sintático. Para tal, as regras de produção criadas para especificar este analisador devem ser entregues a um gerador, compatível com o YACC. Cada uma destas regras deve seguir um dos padrões apresentados a seguir:

```
nao_terminal : componente_1 ... componente_n
               { $$ = new NaoTerminal1 ( $1, ..., $n ); }
               | ...

nao_terminal1 : nao_terminal2
                { $$ = $1; }
                | ...
```

A linguagem de programação orientada a objetos C++ [Str86], usada nas implementações realizadas, permite que estes padrões sejam implementados diretamente. Usando esta linguagem, no primeiro caso apresentado acima, cada ação semântica do analisador sintático deve ser definida como um `new` seguido pelo construtor da respectiva classe.

Explorando ainda mais as características desta linguagem na implementação dos métodos derivados de esquemas de tradução, percebe-se que esta tarefa se resumiria a tomar o texto com a especificação do esquema de tradção e substituir os nomes de símbolo pelos nomes das classes que os representam, precedido da declaração `new`. Nos casos não resolvidos pelo polimorfismo da linguagem, é necessário também criar um trecho de código que tome a decisão sobre qual esquema de tradução aplicar (a equação `TQ` é um bom exemplo deste caso). A interface de uma das classes usadas no nosso último exemplo pode ser a apresentada na figura 2.5.

Em termos organizacionais, cada compilador implementado usando este método foi dividido em 5 partes (cada uma em um arquivo fonte separado): a interface das classes que representam os símbolos da linguagem, a especificação do analisador léxico, a especificação do analisador sintático, a implementação dos métodos que realizam a tradução, e a implementação dos outros métodos de cada classe.

```
class ListComprehension : public DefinedList {
   ExpressionList *qualifier;

   public:
   ListComprehension (ListContentsList *c, ExpressionList *q);
   int GetId (void) { return ListComprehensionId; }
   void Save (FILE *f);
   int Print (FILE *f, int indent, char *sep);
   static ProgramSymbol *Load (FILE *f);

   LambdaExpr* TE (void);
   LambdaExpr* TQ (LambdaExpr **L);
};
```

Figura 2.5: **Interface da classe** `ListComprehension`

Capítulo 3

O desenvolvimento de um compilador para MIRANDA

> *"Miranda is a functional programming language and it is chosen here because it is the best available such language. Arguably, it is the best programming language of any sort at this time".*
>
> *Antoni Diller*

Miranda marcou a época em que foi desenhada como uma linguagem de características espetaculares, por aliar clareza e simplicidade a um alto poder expressivo [Tur86]. Haskell [Hud89, JW91], A [MMS91] e outras foram criadas com base nas características definidas inicialmente nesta linguagem, tais como o estilo equacional de programação, a definição de listas através da especificação de propriedades sobre elementos, o casamento de padrões, entre outras.

A implementação de um compilador para Miranda despertou tanta ou mais atenção que a própria linguagem, pois demandou a criação de um conjunto de algoritmos de tradução, formalizados posteriormente de maneira bastante elegante sob a forma de esquemas de tradução [Jon87]. Entretanto, tais algorítmos são complexos, e as implementações realizadas acabavam requerendo um entendimento profundo do problema. Neste capítulo discutiremos como uma implementação para este compilador foi realizada, usando o método descrito no capítulo anterior.

No desenvolvimento deste compilador foram usadas as especificações propostas por Peyton-Jones [Jon87], compostas por um conjunto de esquemas de tradução e por algumas descrições informais. Estas especificações foram formalizadas, e neste trabalho serão apresentados apenas os esquemas de tradução que foram alterados ou que não aparecem nesta referência. A especificação da sintaxe das linguagens envolvidas é apresentada no apêndice.

No processo de compilação descrito neste trabalho, programas sofrem traduções sucessivas, sendo determinado a cada etapa um programa equivalente ao original escrito em outra linguagem. A partir de um programa em Miranda, expressões lambda são derivadas, e usadas na criação de um conjunto de combinadores. Ao final do processo é obtido um programa escrito em uma pseudo-linguagem de máquina, que pode ser executado diretamente por um interpretador apropriado.

A nossa implementação se utiliza de objetos para representar as linguagens envolvidas no processo de tradução, que são instâncias da classe *Language* descrita no capítulo anterior: **Miranda**, **Lamb**, **Comb** e **Gcode**. A tradução de um programa entre cada par destas linguagens é feita por objetos que são instâncias da classe *Compiler* de nosso *framework*. Os programas resultantes, escritos em **Gcode**, podem ser entregues a uma implementação de máquina G (**GMachine**) para execução, esta uma instância de *Interpreter*.

A organização do compilador é apresentada na figura 3.1. Nesta podemos observar que programas no processo de compilação podem estar representados de duas formas: textual ou em árvore de objetos. Uma destas representações pode ser obtida a partir de um programa escrito em qualquer das linguagens envolvidas, através de traduções sucessivas. A conversão do primeiro para o segundo tipo de representação corresponde à análise sintática do texto do programa, enquanto que o contrário corresponde à impressão formatada (*pretty-print*) da respectiva árvore. Estes aspectos fazem de nossa implementação uma ferramenta de compilação mais abrangente que apenas um compilador de Miranda para Gcode.

3.1 MIRANDA

Miranda é uma linguagem que dá um tratamento especial a tipos de dados. Embora não sejam obrigatórias declarações sobre os tipos de cada expressão, a linguagem é fortemente

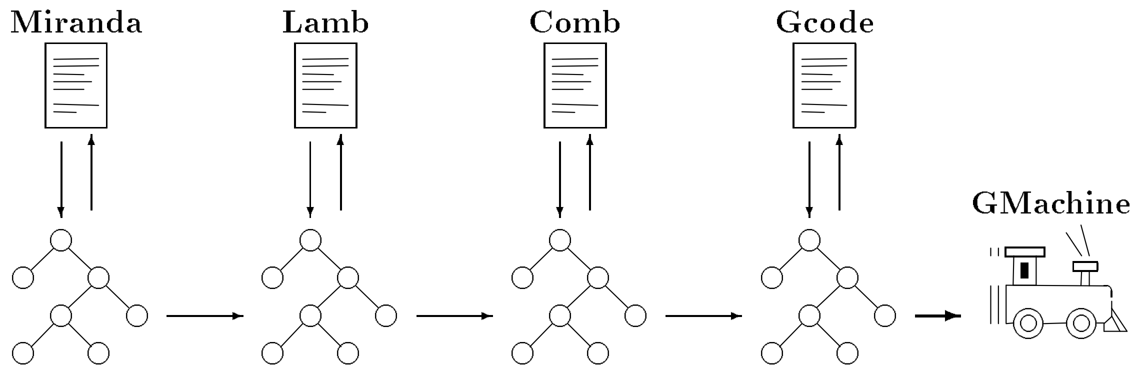

Figura 3.1: **Visão organizacional do compilador**

tipada, e permite a definição de tipos de dados polimórficos e de tipos de dados abstratos [Tur86]. Todas as características relacionadas a tipos não serão abordadas neste trabalho, mas no próximo capítulo será apresentada uma maneira clara para integra-lás à implementação realizada.

Os tipos de dados básicos da linguagem são inteiros, caracteres e booleanos. Os tipos de dados estruturados pré-definidos são listas, tuplas e estruturas (registros), que podem ter componentes de qualquer tipo da linguagem. Listas podem ser definidas das seguintes formas:

1. Propriedades dos elementos: `[x | x > 10 /\ x % 2 = 0]`
2. Enumeração explícita ou intervalo: `[1,2,3]` ou `[1..3]`
3. Seqüências de caracteres (strings): `"isto e' um string"`
4. Cabeça e cauda: `1:[2,3]`
5. Lista vazia: `[]`

Cada programa em Miranda é chamado *script* e é composto por declarações que definem funções, seus domínios e a forma como são obtidos os valores da imagem. Declarações são compostas por conjuntos de equações, e cada uma destas define por casos como é computada a imagem da função. Um típico programa escrito em Miranda é apresentado na figura 3.2.

```
norm (x,y) = sqr ((sq(x) + sq(y)) / 2)

fat 0 = 1
fat n = n * fat (n - 1), n > 0

mod x = norm (x,x)
main = fat (mod (-4))
```

Figura 3.2: **Programa exemplo escrito na linguagem Miranda**

Após o símbolo de definição (`=`) em cada equação aparecem expressões "guardadas". Estas são compostas por uma expressão simples e uma condição (chamada guarda). As primeiras só serão usadas para computar o valor da função caso o respectivo guarda seja avaliado como verdadeiro. No exemplo anterior, `n > 0` é o guarda de `n * fat (n - 1)`.

Os objetos que aparecem do lado esquerdo da definição de cada equação (a exceção do próprio nome da função) são chamados padrões, que devem casar com os parâmetros de chamada para que a equação seja usada para computar o valor da função. O casamento de padões é uma das características mais importantes da linguagem e será detalhado quando for explicada a sua compilação.

Toda a linguagem se baseia na avaliação preguiçosa das expressões, que consiste em avaliá-las apenas quando necessário, da esquerda para a direita. Por exemplo, na expressão `x > 10 /\ x % 2 = 0`, se o valor de `x` for menor que `10`, `x % 2` não será avaliado. Esta característica permite que objetos infinitos sejam manipulados (como listas definidas por intervalos abertos), pois são avaliados apenas até o ponto necessário para que a computação continue.

3.2 Lamb

A utilização de linguagens intermediárias no processo de compilação é comum na maioria das implementações de compiladores para linguagens funcionais. Em [Jon87], [JL92], [Jon92], [MMS91] e [Rot92] uma linguagem que representa termos do *lambda calculus* é utilizada, com as seguintes finalidades:

- representar de maneira clara a semântica da linguagem original em função da linguagem intermediária,
- modularizar e tornar mais inteligível a implementação do compilador, e
- permitir que sejam implementados compiladores para outras linguagens funcionais com base na mesma linguagem intermediária.

Neste trabalho, a utilização de linguagens no processo de compilação recebeu um tratamento mais sistemático, sendo estas representadas de maneira uniforme (estrutura de classes) e compartimentada (estruturas independentes para cada linguagem). Para facilitar esta sistematização, sem deixar de alcançar os objetivos colocados acima, as linguagens intermediárias foram definidas de forma minimal. Estas linguagens são Lamb, apresentada nesta seção, e Comb, discutida posteriormente.

Lamb é uma extensão do *lambda calculus*, ou *enriched lambda calculus* usando a denominação de outros autores, que permite a definição de variáveis, aplicações e abstrações de termos, presentes na teoria original, e ainda declarações com escopo definido, através de `let`s ou `letrec`s.

Ainda nesta linguagem existem termos que terão seu significado avaliado como funções ou constantes em outras etapas do processo de compilação, chamados códigos. Isto é um fator que diferencia esta linguagem das utilizadas por outros autores. Além desta diferença, nas outras linguagens intermediárias existe a declaração `case`, que permite a definição de seleções entre expressões a partir do valor de uma outra expressão. Consideramos, entretanto, que isto feriria a restrição de minimalidade da linguagem, sem que o ganho em clareza fosse significativo. Dessa forma tratamos a seleção de expressões como um código (`CASE`) como outro qualquer.

A representação textual do programa apresentado na seção anterior, escrito agora na linguagem Lamb, é apresentado na figura 3.3.

```
let norm = ^_2.let _5 = (((SELECT (1, 2)) (1, 2)) _2)
                in let _4 = (((SELECT (1, 2)) (1, 1)) _2)
                    in ((FATBAR (sqr ((/ ((+ (sq _4)) (sq _5))) (1, 2))))
                                FAIL)
 in letrec fat = ^_0.(((IF (~ ((== _0) (1, 0))))
                       ((* _0) (fat ((- _0) (1, 1)))))
                       (((IF ((== _0) (1, 0)))
                             (1, 1))
                             FAIL))
        in let mod = ^_1.(norm (((0, 2) _1) _1))
            in (fat (mod (- (1, 4))))
```

Figura 3.3: **Programa exemplo escrito na linguagem Lamb**

Na figura 3.3 a abstração é representada pelo símbolo `^` seguido de uma váriavel, um separador (`.`) e uma expressão. Note que as variáveis do programa são apenas aquelas introduzidas por definições ou abstrações. Além disso, pode-se observar que definições não recursivas são feitas através de `lets`, enquanto que as recursivas através de `letrecs`, e isto é o que ocorre com `norm` e `fat` respectivamente. Ainda na figura aparecem alguns códigos, representados em maiúsculas, que são `SELECT` (seleção de componente de um objeto estruturado), `FATBAT` (avaliação de expressões até que seja encotrada uma que termine sem falha) e `FAIL`.

A representação de valores de dados e seus tipos é feita nesta e nas outras etapas subsequentes do processo de compilação através de pares, onde o primeiro elemento se refere ao tipo do valor, e o segundo ao valor propriamente dito. No exemplo acima `(1,1)` representa o inteiro `1`.

Tipos de dados estruturados são definidos em Lamb pela aplicação de um construtor sobre os componentes da estrutura. Assim, a regra descrita acima vale também para construtores de tipos. Esta escolha evita o que ocorre em muitas implementações, segundo Peyton-Jones [Jon92]: "Compiladores implementam em alguns casos os tipos pré-definidos (listas, tuplas, números) de formas especialmente mágicas, e o programador paga uma penalidade de performance por tipos definidos pelo usuário". Definimos então um tipo de dados para construtores de tipos de dados, representando-o da mesma forma descrita acima: o primeiro componente da representação diz que este objeto é do tipo construtor e o segundo diz qual o tipo dos objetos construidos. O par `(0,2)` que aparece na definição de `norm` no exemplo anterior representa um construtor (tipo `0`), de tuplas (valor `2`).

3.3 A compilação de Miranda para Lamb

A tradução para Lamb de programas escritos em Miranda é feita pela implementação de esquemas e funções de tradução. Exemplos destes são os esquemas (**TE** e **TQ**) e a função **Match**, apresentados no capítulo anterior.

A compilação se inicia ao ser ativado o método do objeto de nível mais alto na representação em árvore do programa fonte, que representa o símbolo *statement_list*. Tal método implementa o seguinte esquema de tradução:

$$\mathbf{TF} : \mathit{statement\_list} \rightarrow \mathit{lambda\_expr}$$
$$\mathbf{TF}\left[\!\!\left[\begin{array}{l}\mathit{statement\_list1}\\ \mathbf{main} = \mathit{guard\_list}\ \mathbf{where}\ \mathit{statement\_list2}\\ \end{array}\right]\!\!\right] \triangleq$$
$$\mathbf{TDA}[\![.]\!](\mathbf{TS}[\![\mathit{statement\_list1}]\!],\ \mathbf{TR}[\![\mathit{guard\_list}]\!](\mathbf{TF}[\![\mathit{statement\_list2}]\!]))$$

A partir daí, os métodos que implementam cada esquema de tradução são ativados, até que as folhas da árvore sejam alcançadas. Em alguns casos, implementações de funções se intercalam a estas ativações, como é o caso de **TDA** no exemplo acima ou **Match** no exemplo do capítulo anterior.

A aplicação de funções se torna a parte não trivial nesta etapa de compilação, por não possuirem uma regra exata a ser seguida quanto a sua aplicação, como ocorre com os esquemas. Passamos a descrever então as principais funções envolvidas.

3.3.1 Remoção de padrões constantes e criação de novas equações condicionais

Miranda permite que uma função definida por casos seja implementada de duas formas diferentes: através de expressões guardadas ou de um conjunto de equações que tenham constantes como padrões. Um exemplo típico é a função fatorial, que pode ter como implementação

```
fat 0 = 1
fat n = n * fat (n - 1)
```

ou

```
fat n = 0, n = 1
      = n * fat (n - 1), otherwise
```

É necessário que estas implementações tenham o mesmo significado, pois representam a mesma função. Assim, estabeleceu-se uma forma canônica para a qual todas as funções definidas por casos devem ser traduzidas. A segunda forma, sem padões constantes, foi escolhida, já que esta restrição é necessária também para que algorítmo para casamento de padrões descrito neste trabalho seja utilizado.

Padrões constantes são substituidos por equações guardadas seguindo os passos:

1. Cria-se um conjunto de variáveis (nomes ainda não utilizados no programa) com cardinalidade igual à da função em questão;
2. Cada padrão constante é substituido pela respectiva variável, acrescentando-se em todas as expressões guardadas da equação em questão uma condição, impondo que a variável seja igual à constante;
3. Se existirem padrões estruturados na equação, substitui-se as constantes que aparecem entre os componentes, aplicando-se o mesmo processo de remoção sobre seus componentes (começando pelo passo 1 e assumindo que a cardinalidade da função é igual à do padrão estruturado);

4. Se todos os padrões da equação tiverem sido substituidos, elimina-se a equação, acrescentando-se a expressão guardada desta em todas as equações remanescentes que definem a mesma função, se estas existirem. Substitui-se em cada uma destas equações todos os padrões pelas respectivas variáveis.

Tomando a primeira implementação para função fatorial, o processo de remoção analisa cada das duas equações. Supondo que {`n`} seja o conjunto de variáveis a ser usado, na primeira equação ocorre a substituição de `0` por `n`, e é acrescida a esta a condição `n = 0`, enquanto que a segunda não se altera. Pela aplicação do quarto passo do processo de remoção, a primeira equação é eliminada, já que teve seu único parâmetro substituido, e sua expressão guardada é transportada para a segunda. Após estes passos, a primeira implementação de fatorial exibida é transformada na segunda.

O esquema **TRPC**, apresentado na figura 3.4, define formalmente os três primeiros passos. Um esquema complementar, não apresentado aqui, define formalmente o quarto passo. É importante lembrar que tal esquema deve ser aplicado a padrões e expressões guardadas já traduzidas para Lamb.

3.3.2 Compilação do casamento de padrões

O casamento de padrões corresponde à verificação se os parâmetros atuais, que serão usados no cômputo de uma função, têm o mesmo formato dos parâmetros formais de alguma equação que define esta função. Com base nesta escolha é definida, em tempo de execução, a equação que será usada.

Duas definições para a criação de código que execute esta função podem ser encontradas na literatura: uma menos eficiente, mas usada na maioria das linguagens funcionais, traduz o programa para expressões condicionais, com `cases` e `ifs` [Jon87], e outra mais complexa, usada em ML, traduz o programa para árvores de decisão [Rot92]. Optamos neste trabalho pela primeira, devido à sua clareza e à possibilidade de aplicar transformações posteriores que a tornam mais eficiente. Ambos algoritmos tomam como entrada as definições de função presentes no programa.

TRPC : *lambda_expr_list* $\rightarrow$ *lambda_expr* $\times$ *lambda_expr_list* $\times$ *boolean*

TRPC[*var_node, lambda_expr_list*]

$\triangleq$ *(, var_node* **TRPC**[*lambda_expr_list*], FALSE*)*

TRPC[*code_node, lambda_expr_list*]

$\triangleq$ *((*IF *((*EQ *var_node) code_node)),*
var_node **TRPC**[*lambda_expr_list*], TRUE*)*

TRPC[*val_node, lambda_expr_list*]

$\triangleq$ *((*IF *((*EQ *var_node) val_node)),*
var_node **TRPC**[*lambda_expr_list*], TRUE*),*
se **não IsConstructor**[.]*(val_node)*

$\triangleq$ *(, val_node* **TRPC**[*lambda_expr_list*], FALSE*),*
senão

TRPC[*(lambda_expr1 lambda_expr2) lambda_expr_list*]

$\triangleq$ *(lambda_expr3* AND *lambda_expr5,*
(var_node **TRPC**[*lambda_expr_list*]*),* TRUE*),*
se boolean1 e boolean2

$\triangleq$ *(lambda_expr3,*
((lambda_expr4 lambda_expr2) **TRPC**[*lambda_expr_list2*]*),* FALSE*),*
se boolean1

$\triangleq$ *(lambda_expr5,*
((lambda_expr1, lambda_expr6) **TRPC**[*lambda_expr_list2*]*),* FALSE*),*
se boolean2

$\triangleq$ *(, (lambda_expr1 lambda_expr2)* **TRPC**[*lambda_expr_list2*]*),* FALSE*),*
senão

onde (lambda_expr3, lambda_expr4, boolean1) = **TRPC**[*lambda_expr1*],
(lambda_expr5, lambda_expr6, boolean2) = **TRPC**[*lambad_expr2*]

Figura 3.4: **Esquema de tradução TRPC**

No exemplo anterior, se `norm` fosse definida como

```
norm (x,y) = sqr ((sq(x) + sq(y)) / 2)
norm x     = error x
```

o algorítmo de tradução utilizado se comportaria da maneira apresentada a seguir:

1. Cria-se um conjunto de variáveis artificiais, que será usado para substituir as variáveis usadas no programa. Este conjunto deve ser o mesmo usado na remoção de padrões constantes. Como `norm` só possui um argumento, {_2} poderia ser tal conjunto.

2. Divide-se as equações em grupos, de acordo com o formato do primeiro padrão:

```
norm (x,y) = sqr ((sq(x) + sq(y)) / 2) <-- Forma um grupo (construtor)
norm x     = error x                   <-- Forma outro grupo (variavel)
```

3. Para grupos nos quais o primeiro padrão é um construtor, aplica-se o algorítmo sobre os componentes da estrutura, usando um novo conjunto de variáveis artificiais, com cardinalidade igual ao número de componentes da estrutura. No exemplo, este conjunto será {_4, _5}. A expressão resultante deste passo será aninhada em um conjunto de **let**s, onde cada definição é uma seleção de um componente da estrutura. Obtem-se:

```
letrec _5 = (((SELECT (1, 2)) (1, 2)) _2)
       _4 = (((SELECT (1, 2)) (1, 1)) _2)
    in sqr ((sq(x) + sq(y)) / 2)
```

4. Termina-se a tradução do primeiro grupo da seguinte maneira:

```
letrec _5 = (((SELECT (1, 2)) (1, 2)) _2)
       _4 = (((SELECT (1, 2)) (1, 1)) _2)
    in ((FATBAR (sqr ((/ ((+ (sq _4)) (sq _5))) (1, 2))))
              FAIL)
```

5. Em grupos de equações onde os primeiros padrões são variáveis, estas são substituidas pela respectiva variável artificial, aplicando o algorítmo sobre o corpo da definição. No exemplo, `x` é substituido por _2, e o algoritmo é aplicado sobre `error x`, resultando em:

```
(((FATBAT (error _2)) FAIL)
```

6. Une-se os resultados da tradução dos grupos sob uma abstracão de todas as variáveis artificiais criadas para a tradução da equação:

```
^_2.((CASE (letrec _5 = (((SELECT (1, 2)) (1, 2)) _2)
                   _4 = (((SELECT (1, 2)) (1, 1)) _2)
                 in ((FATBAR (sqr ((/ ((+ (sq _4)) (sq _5))) (1, 2))))
                             FAIL)))
           (((FATBAR (error _2)) FAIL)))
```

7. Nos casos onde não há mais expressões a traduzir, o resultado da tradução é composto pela aplicação de FATBAR aos resultados dos passos anteriores, e esta expressão aplicada a FAIL.

Uma definição mais rigorosa para tal algoritmo pode ser encontrada em [Jon87].

É importante observar que o algoritmo apresentado nesta seção só irá se comportar corretamente se os padrões constantes forem removidos, como descrito na seção anterior.

3.3.3 Análise de Dependências

Chama-se de Análise de Dependências à determinação das inter-dependências entre definições de uma mesma declaração. Utiliza-se normalmente esta informação na criação de um conjunto de declarações aninhadas equivalente, fazendo que o escopo de cada definição seja mínimo, abrangendo todos os pontos onde originalmente é usada. Esta técnica é análoga à Análise de Fluxo de Dados em linguagens de programação imperativas.

Tomando o termo resultante da aplicação do casamento de padrões sobre a função `norm`, definida nos exemplos apresentados nas seções anteriores, tal análise resultaria na seguinte tradução:

```
letrec  _5 = ...
        _4 = ...      ==>   let  5 = ...
    in  ...                 in   let  4 = ...
                                 in   ...
```

Esta tradução pode ser realizada sem alteração na semântica do termo, pois:

1. a definição de `_5` não depende de `_4`;
2. a definição de `_4` não depende de `_5`;
3. de 1 e 2, não é necessário um `letrec` que englobe as duas definições;
4. as definições de `_4` e de `_5` não são recursivas, portanto não precisam ser feitas por meio de `letrecs`, para cada uma delas;
5. de 1 e 2, não importa a ordem em que estas declarações estejam aninhadas.

Os esquemas de tradução que definem esta análise são os apresentados na figura 3.5. Nestes esquemas, a função semântica **FreeIn** é usada na verificação se uma variável aparece livre em uma expressão.

Nos limitamos a enunciar o esquema **TDA** para declarações com uma e duas definições, respectivamente. Para conjuntos maiores de definições, a quantidade de testes necessários para constatar se uma definição recursiva é necessária (usando **letrec**) cresce exponencialmente. Este fato caracteriza a análise de dependências como uma de natureza eminentemente combinatória. Normalmente são usados nas implementações deste esquema algorítmos sobre grafos.

Na implementação realizada, um grafo é construido, tendo como nós as definições e como arestas as inter-dependências presentes na declaração original. Através de um algorítmo que determina as componentes fortemente conexas deste grafo, cria-se um hi-grafo (onde os nós podem conter grafos ou sub-grafos). Neste hi-grafo, os conjuntos de definições com dependência mútua do grafo original são agrupados como componentes de um nó, e assim as arestas passam a representar dependências entre conjuntos de definições. Nós com apenas um componente são usados para criar `lets`, enquanto que os demais são usados para

$\mathbf{TDA} : \mathit{lambda\_expr\_list} \times \mathit{lambda\_expr} \rightarrow \mathit{lambda\_expr}$

$\mathbf{TDA}[.]$*(var_node = lambda_expr1, lambda_expr2)*

$\triangleq$ **let** *var_node = lambda_expr1* **in**

lambda_expr2, se $\mathbf{FreeIn}[.]$*(var_node, lambda_expr2)*

$\triangleq$ *lambda_expr2, senão*

$\mathbf{TDA}[.]$*(var_node1 = lambda_expr1,*

var_node2 = lambda_expr2,

lambda_expr3)

$\triangleq$ **letrec** *var_node1 = lambda_expr1,*

var_node2 = lambda_expr2 **in**

lambda_expr3,

se $\mathbf{FreeIn}[.]$*(var_node2, lambda_expr1) e*

$\mathbf{FreeIn}[.]$*(var_node1, lambda_expr2) e*

($\mathbf{FreeIn}[.]$*(var_node1, lambda_expr3) ou*

$\mathbf{FreeIn}[.]$*(var_node2, lambda_expr3))*

$\triangleq$ **let** *var_node1 = lambda_expr1* **in**

$\mathbf{TDA}[.]$*(var_node2 = lambda_expr2, lambda_expr3),*

se $\mathbf{FreeIn}[.]$*(var_node1, lambda_expr2) e*

$\mathbf{FreeIn}[.]$*(var_node2, lambda_expr3)*

$\triangleq$ $\mathbf{TDA}[.]$*(var_node2 = lambda_expr2, lambda_expr3), senão*

Figura 3.5: **Esquema de tradução TDA**

criar `letrecs`. As arestas do hi-grafo são usadas para determinar como as declarações serão aninhadas.

Escolheu-se implementar este procedimento como uma busca *depth-first* pela solução, usando como entrada um objeto da classe *Graph* e apresentando como saida outro da classe *HiGraph*, ambas pertencentes à estrutura de classes para apoio ao desenvolvimento de compiladores apresentada no capítulo anterior.

Comparando nossa implementação com outras já realizadas, observa-se que:

- Normalmente esta análise é feita após a compilação de todo o programa, sendo necessária uma busca pelas declarações sobre toda a representação criada, para realizar as traduções. Como estas traduções são feitas apenas sobre `lets` e `letrecs`, não tendo efeito sobre outros termos, optamos por realizá-las no instante em que os objetos que representam estes símbolos são criados, eliminando a necessidade de buscas. Portanto, em nossa implementação nunca é encontrada uma declaração `new LetExpr (a,b)` ou `new LetrecExpr (a,b)`, mas sim `TDA (a, b)`;
- A primeira regra do esquema permite que definições não usadas sejam eliminadas, já que não existem em linguagens funcionais puras desvios de controle ou ponteiros que obriguem a avaliação destas expressões.

Embora um tanto óbvias, tais otimizações não foram propostas em nenhuma das implementações discutidas em [MMS91], [Jon87] ou [JL92].

Além dos benefícios já mencionados, a realização deste tipo de análise nesta etapa da compilação é necessária para que outros esquemas de tradução, como o *lambda lifting*, possam ser aplicados [Jon87].

3.4 Comb

Para que seja possível executar as expressões da linguagem Lamb, uma das seguintes técnicas pode ser adotada:

- interpretação: equivale a percorrer o corpo da expressão, instanciando as variáveis e substituindo-as pelos parâmetros formais, realizando reduções quando possível (*template instantiation*), ou
- compilação: gerar código que irá criar em tempo de execução o corpo da expressão.

Template instantiation é uma técnica de interpretação que apresenta o sério inconveniente de ter que percorrer o corpo da expressão lambda, substituindo variáveis pelos seus valores já instanciados. Este percurso é implementado como um caminhamento em uma árvore, substituindo as folhas que representam as variáveis amarradas na expressão pelos respectivos valores. Posteriormente reduções são realizadas, substituindo nós que representam expressões pelos valores correspondentes computados. Para que uma redução possa ser feita, todo o corpo da expressão já deve ter sido instanciado.

Se o corpo de uma expressão não possuir variáveis livres, no momento em que suas variáveis forem instanciadas, a expressão pode ser reduzida. Explorando esta possibilidade, a técnica de compilação *lambda lifting* abstrai as variáveis livres de cada expressão, compilando-a num código que irá construir sua instância quando executado. Esta construção já é feita com os parâmetros atuais da função substituindo seus parâmetros formais. Neste processo, a geração de código é simples, e este é mais eficiente que a interpretação.

As expressões criadas pelo *lambda lifting* são chamadas supercombinadores. Usando a definição apresentada por Peyton-Jones, "um supercombinador $S é uma expressão lambda na forma $\lambda x1.\lambda x2.\lambda x3....\lambda xn.E$ onde E não é uma abstração, tal que $S não tem variáveis livres, cada expressão em E é um supercombinador e $n \geq 0$ (não precisam aparecer lambdas)" [Jon87].

De acordo com a definição acima, podemos representar um supercombinador da forma `$S x1 x2 ... xn = E`, eliminando assim a necessidade de representar abstrações. Estamos aptos então a criar outra linguagem semelhante a Lamb, permitindo que nesta apareçam definições de supercombinadores e eliminando as abstrações. Esta linguagem chamamos de Comb.

```
$2 _1 = ($3 (((0, 2) _1) _1))
$1 _0 = (((IF (~ ((== _0) (1, 0))))
              ((* _0) ($1 ((- _0) (1, 1)))))
              (((IF ((== _0) (1, 0)))
                    (1, 1))
                    FAIL))
$3  = (($0 ((SELECT (1, 2)) (1, 2))) ((SELECT (1, 2)) (1, 1)))
$0 _2, #0, #1 = let _5 = (#0 _2)
                in let _4 = (#1 _2)
                    in ((FATBAR (sqr ((/ ((+ (sq _4)) (sq _5))) (1, 2))))
                                  FAIL)
main  = ($1 ($2 (- (1, 4))))
```

Figura 3.6: **Programa exemplo escrito na linguagem Comb**

3.5 A compilação de Lamb para Comb

Em Comb, o programa apresentado nas seções anteriores seria escrito como na figura 3.6.

No processo de compilação deste programa, de Lamb para Comb, são aplicados os métodos que implementam os esquemas correspondentes ao *float-out let(recs)* e ao *lambda lifting* nesta ordem. Como o primeiro esquema de compilação só é necessário devido a uma otimização no segundo, apresentaremos estes esquemas na ordem oposta à que são aplicados.

3.5.1 *Lambda Lifting*

Lambda lifting é a técnica usada para remover variáveis livres e abstrações de expressões em linguagens semelhantes ao *lambda calculus*, criando definições globais chamadas combinadores. Uma boa definção para esta técnica é apresentada por Meira em [MMS91]:

- para cada variável livre em uma definição de função local, adicionamos um argumento à função (os nomes de funções são tratados como constantes e não são abstraídos).

- aplicamos as mesmas variáveis livres a cada aplicação da função, substituindo `f` por `f` $x1...xn$, onde $x1...xn$ é o conjunto de variáveis livres na definição de `f`.

Na figura 3.7 são apresentados os esquemas de tradução correspondentes à esta técnica. Na figura podemos ver pela assinatura do esquema que o resultado de sua aplicação resulta em uma lista de combinadores (definições levadas para o nível global), e em uma expressão que será usada para substituir a expressão original.

Esta técnica está fundamentada na abstração das variáveis livres de cada expressão. Incrementando esta idéia, pode-se pensar em abstrair expressões inteiras que não contém variáveis que estão amarradas às definições de funções. Isto é chamado *fully lazy lambda lifting*.

Um *lambda lifting* totalmente preguiçoso consiste em determinar as expressões livres maximais que aparecem em cada definição de função, abstraindo-as. Esta otimização permite que se economize tempo e memória durante a execução, avaliando-se cada expressão o menor número de vezes possível.

O exemplo da figura 3.6 mostra que, ao ser aplicada esta técnica, à abstração `^_2.lambda_expr`, esta não tem abstraida de seu corpo apenas `_2`, mas também `((SELECT (1, 2)) (1, 2)))` e `((SELECT (1, 2)) (1, 1)))`, que são expressões sem variáveis livres, e portanto idenpendentes.

A determinação das expressões maximais está baseada no cálculo do número de nível de cada expressão, como apresentado na próxima seção.

3.5.2 *Float-Out let(rec)s*

Para evitar que o corpo das definições que aparecem em `lets` e `letrecs` tenha que ser recomputado a cada uso da expressão, definiu-se a técnica chamada *float-out let(rec)s*. Esta impede a perda da característica totalmente preguiçosa das expressões onde este cálculo

$\mathbf{TLL} : \mathit{lambda\_expr} \rightarrow \mathit{combinator\_list} \times \mathit{lambda\_expr}$

$\mathbf{TLL}[\![\mathit{(lambda\_expr1\ lambda\_expr2)}]\!]$

$\quad\triangleq\quad \mathit{(combinator\_list1\ combinator\_list2,\ (lambda\_expr1\ lambda\_expr2))}$

$\qquad \mathit{onde\ (combinator\_list1,\ lambda\_expr1)} = \mathbf{TLL}[\![\mathit{lambda\_expr1}]\!]$

$\qquad\quad \mathit{(combinator\_list2,\ lambda\_expr2)} = \mathbf{TLL}[\![\mathit{lambda\_expr2}]\!]$

$\mathbf{TLL}[\![\mathrm{LAMBDA}\ \mathit{var\_node}\ \mathrm{DOT}\ \mathit{lambda\_expr}]\!]$

$\quad\triangleq\quad \mathit{(var\_node1\ var\_node} = \mathit{lambda\_expr\ combinator\_list,\ var\_node1)}$

$\qquad \mathit{onde\ (combinator\_list,\ lambda\_expr1)} = \mathbf{TLL}[\![\mathit{lambda\_expr}]\!]$

$\mathbf{TLL}[\![\mathbf{let}\ \mathit{var\_node} = \mathit{lambda\_expr1}\ \mathbf{in}\ \mathit{lambda\_expr2}]\!]$

$\quad\triangleq\quad \mathit{(var\_node} = \mathit{lambda\_expr1\ combinator\_list,\ lambda\_expr3)}$

$\qquad\quad \mathit{se\ não\ existe\ var\_node\ tal\ que}\ \mathbf{FreeIn}[\![.]\!]\mathit{(var\_node,\ lambda\_expr1)}$

$\qquad \mathit{onde\ (combinator\_list,\ lambda\_expr3)} = \mathbf{TLL}[\![\mathit{lambda\_expr2}]\!]$

$$\mathbf{TLL}\left[\!\!\left[\begin{array}{l}\mathbf{letrec}\ \mathit{var\_node} = \mathit{lambda\_expr1,}\\ \qquad \mathit{lambda\_expr\_list1}\\ \mathbf{in}\ \mathit{lambda\_expr2}\end{array}\right]\!\!\right]$$

$\quad\triangleq\quad \mathit{(var\_node} = \mathit{lambda\_expr1\ combinator\_list,\ lambda\_expr3)}$

$\qquad\quad \mathit{se\ não\ existe\ var\_node\ tal\ que}\ \mathbf{FreeIn}[\![.]\!]\mathit{(var\_node,\ lambda\_expr1)}$

$\qquad \mathit{onde\ (combinator\_list,\ lambda\_expr3)} = \mathbf{TLL}[\![\mathit{lambda\_expr2}]\!]$

$\mathbf{TLL}[\![\mathit{lambda\_expr\ lambda\_expr\_list}]\!]$

$\quad\triangleq\quad \mathit{(combinator\_list1\ combinator\_list2,\ lambda\_expr1\ lambda\_expr2)}$

$\qquad \mathit{onde\ (combinator\_list1,\ lambda\_expr1)} = \mathbf{TLL}[\![\mathit{lambda\_expr}]\!]$

$\qquad\quad \mathit{(combinator\_list2,\ lambda\_expr2)} = \mathbf{TLL}[\![\mathit{lambda\_expr\_list}]\!]$

$\mathbf{TLL}[\![\mathit{lambda\_expr}]\!] \triangleq \mathit{(,\ lambda\_expr)}$

Figura 3.7: **Esquema de tradução TLL**

não é sempre necessário. O *float-out* equivale a retirar declarações do corpo das expressões, quando este não depende das variáveis definidas ai. Para determinar as dependências entre expressões maximais, utiliza-se o número de nível de cada usa destas, que indica em quantas definições ou abstrações a expressão esta inclusa.

Tomando a estrutura em árvore que representa cada programa, esta técnica usa o caminhamento abaixo na árvore para calcular o nível das abstrações do programa, e no sentido oposto para calcular o nível das outras expressões e para retirar `lets` e `letrecs` do corpo de cada expressão, se for o caso.

Para cada termo da linguagem Lamb, uma regra é usada na determinação de seu nível:

- variáveis: é igual ao número de abstrações nas quais está inclusa sem que esteja livre.
- aplicações: é o maior dos níveis dos componentes.
- abstrações: igual ao nível da variável que limita.
- `lets`: a variável definida possui nível igual ao do corpo de sua definição e este é usado no cálculo do nível do corpo do let. O nível encontrado aí é o da expressão.
- `letrecs`: a regra é similar à usada para `let`, considerando porém como nível de todas as variáveis que aparecem definidas o maior dos níveis dos corpos de suas definições.
- outros: 0

No exemplo apresentado anteriormente,

```
^_2.let _5 = (((SELECT (1, 2)) (1, 2)) _2)
     in let _4 = (((SELECT (1, 2)) (1, 1)) _2)
         in ((FATBAR (sqr ((/ ((+ (sq _4)) (sq _5))) (1, 2))))
                      FAIL)
```

o nível de `_2` é 1. Assim, `_5` e `6` também são associados a este valor na avaliação do corpo do `let` mais interno. Neste só aparecem aplicações onde os componentes tem nível menor

$$\mathbf{TFO} : \mathit{lambda\_expr} \rightarrow \mathit{lambda\_expr}$$

$$\mathbf{TFO}[\![((\mathbf{let}\ \mathit{var\_node} = \mathit{lambda\_expr1}\ \mathbf{in}\ \mathit{lambda\_expr2})\ \mathit{lambda\_expr3})]\!]$$
$$\triangleq \quad \mathbf{let}\ \mathit{var\_node} = \mathbf{TFO}[\![\mathit{lambda\_expr1}]\!]\ \mathbf{in}$$
$$(\mathbf{TFO}[\![\mathit{lambda\_expr2}]\!]\ \mathbf{TFO}[\![\mathit{lambda\_expr3}]\!])$$
$$\textit{se não}\ \mathbf{FreeIn}[\![.]\!](\mathit{var\_node}, \mathit{lambda\_expr3})$$

$$\mathbf{TFO}[\![(\mathit{lambda\_expr1}\ (\mathbf{let}\ \mathit{var\_node} = \mathit{lambda\_expr2}\ \mathbf{in}\ \mathit{lambda\_expr3}))]\!]$$
$$\triangleq \quad \mathbf{let}\ \mathit{var\_node} = \mathbf{TFO}[\![\mathit{lambda\_expr2}]\!]\ \mathbf{in}$$
$$(\mathbf{TFO}[\![\mathit{lambda\_expr1}]\!]\ \mathbf{TFO}[\![\mathit{lambda\_expr3}]\!])$$
$$\textit{se não}\ \mathbf{FreeIn}[\![.]\!](\mathit{var\_node}, \mathit{lambda\_expr1})$$

$$\mathbf{TFO}[\![\mathrm{LAMBDA}\ \mathit{var\_node1}\ \mathrm{DOT}\ \mathbf{let}\ \mathit{var\_node2} = \mathit{lambda\_expr2}\ \mathbf{in}\ \mathit{lambda\_expr3}]\!]$$
$$\triangleq \quad \mathbf{let}\ \mathit{var\_node2} = \mathbf{TFO}[\![\mathit{lambda\_expr2}]\!]\ \mathbf{in}$$
$$(\mathrm{LAMBDA}\ \mathit{var\_node1}\ \mathrm{DOT}\ \mathbf{TFO}[\![\mathit{lambda\_expr3}]\!])$$

$$\mathbf{TFO}\left[\!\!\left[\begin{array}{l} \mathbf{let}\ \mathit{var\_node1} = \mathbf{let}\ \mathit{var\_node2} = \mathit{lambda\_expr1} \\ \qquad\qquad\qquad\quad \mathbf{in}\ \mathit{lambda\_expr2} \\ \mathbf{in}\ \mathit{lambda\_expr3} \end{array}\right]\!\!\right]$$
$$\triangleq \quad \mathbf{let}\ \mathit{var\_node2} = \mathbf{TFO}[\![\mathit{lambda\_expr1}]\!]$$
$$\mathbf{in}\ \mathbf{let}\ \mathit{var\_node1} = \mathbf{TFO}[\![\mathit{lambda\_expr2}]\!]$$
$$\mathbf{in}\ \mathbf{TFO}[\![\mathit{lambda\_expr3}]\!]))$$
$$\textit{se não}\ \mathbf{FreeIn}[\![.]\!](\mathit{var\_node2}, \mathit{lambda\_expr3})$$

$$\mathbf{TFO}\left[\!\!\left[\begin{array}{l} \mathbf{letrec}\ \mathit{var\_node1} = \mathbf{let}\ \mathit{var\_node2} = \mathit{lambda\_expr1} \\ \qquad\qquad\qquad\qquad \mathbf{in}\ \mathit{lambda\_expr2}, \\ \qquad \mathit{lambda\_expr\_list2} \\ \mathbf{in}\ \mathit{lambda\_expr3} \end{array}\right]\!\!\right]$$
$$\triangleq \quad \mathbf{let}\ \mathit{var\_node2} = \mathbf{TFO}[\![\mathit{lambda\_expr1}]\!]$$
$$\mathbf{in}\ \mathbf{letrec}\ \mathbf{TFO}[\![\mathit{lambda\_expr\_list2}]\!]$$
$$\mathbf{in}\ \mathbf{TFO}[\![\mathit{lambda\_expr3}]\!]$$
$$\textit{se não}\ \mathbf{FreeIn}[\![.]\!](\mathit{var\_node2}, \mathit{lambda\_expr3})$$

$$\mathbf{TFO}[\![\mathit{lambda\_expr}\ \mathit{lambda\_expr\_list}]\!] \triangleq \mathbf{TFO}[\![\mathit{lambda\_expr}]\!]\ \mathbf{TFO}[\![\mathit{lambda\_expr\_list}]\!]$$

$$\mathbf{TFO}[\![\mathit{lambda\_expr}]\!] \triangleq \mathit{lambda\_expr}$$

Existem esquemas análogos para **letrecs**.

Figura 3.8: **Esquema de tradução TFO**

ou igual a 1, sendo este o nível da expressão como um todo. Por apresentar o mesmo nível da abstração, o `lets` mais externo não poderia ser retirados do corpo desta, sendo aplicada a última regra apresentada na figura 3.8.

Se esta técnica for aplicada antes do *lambda lifting*, outra otimização pode ser feita, ordenando os parâmetros de cada supercombinador, do menor para o maior número de nível. Isto permite que sejam eliminados aqueles parâmetros que são variáveis e aparecem do lado direito da aplicação que representa o corpo do combinador, quando isto ocorre. Além disso, devido a aplicação desta otimização, são criados em tempo de execução menos combinadores, e estes maiores que os obtidos anteriormente, porque aqueles que são avaliados mais freqüentemente (os com menor nível) tem este número reduzido pela ordenação.

3.6 Gcode

GCODE é a linguagem de mais baixo nível utilizada neste trabalho. Cada uma de suas instruções define o comportamento de combinadores e como estes manipulam dados dentro da arquitetura de uma máquina de pilha. Uma destas arquiteturas, a máquina G, é apresentada nas próximas seções.

A semântica de GCODE aparece definida na literatura através de dois enfoques distintos:

1. operacional: são usadas transições entre os estados de uma máquina abstrata [Jon87].
2. transformacional: são usados esquemas de tradução para definir como os programas são compilados na linguagem de uma máquina de pilha com menos instruções, chamada TIM (*three instruction machine*) [Jon92].

A segunda proposta é mais recente, não sendo claro até o momento qual dos enfoques é mais vantajoso. Na implementação realizada, escolhemos então o primeiro, por se tratar de uma proposta mais consolidada e documentada.

3.7 A compilação de Comb para Gcode

A compilação de programas para Gcode é realizada por métodos dos objetos que representam símbolos da linguagem Comb. Estes foram implementados seguindo os esquemas de tradução propostos por Peyton-Jones:

- **F**: Compila um combinador, colocando na pilha seus argumentos.
- **R**: Gera código para aplicar o supercombinador a seus argumentos.
- **RS**: Continua a compilação iniciada por **R**.
- **CLetrec**: Constroi instâncias para as expressões definidas em um `letrec`.
- **C**: Constroi o grafo para uma instância de uma expressão.
- **E**: Avalia uma expressão, deixando-a no topo da pilha.
- **ES**: Continua a compilação iniciada por **ES**.
- **B**: Avalia uma expressão, deixando na pilha o valor básico correspondente.

A compilação de cada combinador se inicia pela aplicação do esquema **F**, que cria uma pseudo-instrução (`GLOBALSTART`) para rotular o início do código que implementa tal combinador. Este esquema cria ainda um contexto, que associa cada nome de variável à posição que esta irá ocupar na pilha quando o combinador for executado, contendo todas as variáveis que são parâmetros do combinador.

A partir daí, os outros esquemas são aplicados recursivamente, nos casos em que são necessários. Usando o contexto criado anteriormente, os esquemas realizam a transformação de referências a variáveis nos endereços que estas irão ocupar na pilha durante a execução do combinador. Ainda neste passo, expressões que são códigos (*code_node*) são traduzidas para as respectivas funções pré-definidas. Ao final do processo, a função `$1` apresentada anteriormente seria traduzida para o programa apresentado na figura 3.9.

Os programas gerados em GCODE são compostos por instruções de uma pseudo-linguagem de máquina de pilha. As mais importantes entre estas são aquelas que empilham

```
GLOBALSTART $1 1;
PUSH 0;
EVAL;
GET;
PUSHVAL 1 0;
GET;
EQ;
NEG;
JFALSE L0;
PUSH 0;
EVAL;
GET;
PUSH 1;
EVAL;
GET;
PUSHVAL 1 1;
GET;
SUB;
MKVAL;
PUSHGLOBAL $1;
MKAP 1;
EVAL;
GET;
MUL;
UPDTVAL 2;
POP 1;
RETURN;
LABEL L0;
PUSH 0;
EVAL;
GET;
PUSHVAL 1 0;
GET;
EQ;
JFALSE L1;
PUSHVAL 1 1;
GET;
UPDTVAL 2;
POP 1;
RETURN;
LABEL L1;
POP 1;
JUMP L_ERROR;
```

Figura 3.9: **Função `$1` escrita em GCODE**

apontadores, valores ou combinadores (`PUSH`, `PUSHVAL` e `PUSHGLOBAL`), as que retiram ou atualizam elementos da pilha (`POP`, `UPDATE` e `UPDTVAL`), aquelas que realizam a avaliação de combinadores e controlam a execução destes (`EVAL`, `UNWIND` e `RETURN`), as que transformam tipos de dados (`GET`, `MKVAL` e `MKAP`), aquelas que realizam o controle de execução (`JFALSE` e `JUMP`), as que realizam comparações entre valores (`EQ` e outras) e as que realizam operações aritméticas sobre valores básicos (`NEG` e outras).

Entre todos os combinadores, o único que necessita de um tratamento especial é `main`, que representa a ativação da função principal computada pelo programa. O resultado desta função deverá ser impresso como resultado do programa, após ser avaliado. Este tratamento é feito por um trecho de código adicional, criado como preâmbulo da tradução da definição de `main`. Tal trecho de código é apresentado na figura 3.10.

Nesta figura, podem ser notados buracos preenchidos por instruções não mostradas. No primeiro deles, entre `LABEL L_ERROR` e `JUMP L_END`, deve ser inserido o código para tratamento de erros do programa. Convencionalmente nenhuma instrução é colocada aí, mas isto pode ser feito, como sugerido no último capítulo. Os demais buracos são preenchidos com

```
BEGIN;
PUSHGLOBAL main;
EVAL;
PRINT;
LABEL L_END;
END;
LABEL L_ERROR;
...
JUMP L_END;
...
GLOBALSTART main 0;
...
```

Figura 3.10: **Compilação do combinador `main`**

o resultado da compilação dos outros combinadores do programa e da definição de `main`, respectivamente.

3.8 A máquina G

Redução de grafos é uma técnica que permite o cálculo eficiente de valores de funções. Está baseada na construção, em tempo de execução, de uma instância do corpo do supercombinador que representa a função. Atualmente existem alguns modelos de redução propostos, induzindo a criação de diferentes máquinas que os implementem.

Normalmente são usadas máquinas definidas conceitualmente, chamadas abstratas, que permitem o raciocínio sobre o *back end* do processo de tradução de linguagens funcionais, sem que seja necessário entrar em detalhes sobre a implementação, mas facilitando esta atividade quando necessário, e envolvendo um número mínimo de conceitos. A máquina G é um destes casos.

Embora várias definições para a máquina G existam, como as apresentadas em

[Jon87], [JL92] e [MMS91], todas coincidem em um conjunto mínimo de componentes. Os componentes da máquina abstrata definida por Peyton-Jones na primeira referência, e adotados neste trabalho, são:

- **O**: saída padrão;
- **G**: grafo ou *heap*, contendo todas as células de memória que podem ser usadas durante a execução do programa;
- **S**: pilha, contendo valores básicos ou apontadores para elementos de **G**;
- **C**: programa a ser interpretado;
- **D**: pilha de pares de valores ou *dump*, onde o primeiro componente é uma pilha (proveniente de **S**) e o segundo é um apontador para uma intrução do programa (pertencente a **C**).

Cada elemento do *dump* representa o estado da máquina no instante em que a execução do código de um combinador foi interrompida para que um segundo tivesse seu valor computado. Este estado é recuperado quando a execução do código do segundo combinador se encerrar, sendo colocado no topo da pilha o resultado desta computação. Elementos do *dump* equivalem a ambientes em linguagens de programação convencionais.

A memória de uma GMachine não é acessível diretamente pelas instruções do programa. Estas só se tornam visíveis quando a pilha da máquina é utilizada. A pilha pode conter apenas valores básicos, que são pares compostos por tipo e valor, ambos inteiros, ou apontadores para células do *heap*.

Cada célula do *heap* pode conter valores, nomes de combinadores ou aplicações de uma célula sobre outra. No primeiro e segundo casos, os dados são armazenados da maneira convencional, enquanto no último são criados apontadores para as células do *heap* que compõem a aplicação. Todos estes casos estão sujeitos à coleta de lixo.

Para que seja possível manipular um valor diretamente, é necessário retirá-lo da respectiva célula do *heap*, e recolocá-lo lá depois da operação. Quando repetidas operações são feitas, pode-se manter os valores intermediários na pilha, até que o resultado final seja alcançado, resultando numa otimização considerável quanto ao número de movimentações e ao número de células que não necessitarão ser visitadas pelo coletor de lixo, já que estas sempre estarão referenciadas. Por estas razões, consideramos que a pilha da máquina pode conter valores básicos.

O comportamento da máquina depende da execução das instruções do programa, e cada uma destas tem semântica operacional definida em função de transições de estados. Esta definição pode ser encontrada em [Jon87]. Para exemplificar como um programa é executado, na figura 3.11 apresentamos parte do cálculo do valor da função `$1`, aplicada sobre 4, usada no exemplo das seções anteriores. Na figura, a primeira coluna contém a instrução a ser executada, enquanto que na segunda e terceira aparecem o estado da pilha e do dump após a execução desta.

A figura mostra que, antes da execução do combinador `$1`, a pilha referencia uma aplicação deste sobre seus parâmetros. A instrução `UNWIND` desmonta esta aplicação, colocando os parâmetros na pilha de forma que eles possam ser retirados de lá após a computação do valor do combinador. Várias manipulações dos dados na pilha são realizadas, como cópias de trabalho dos parâmetros atuais e a transformação destes em valores básicos (representados fora dos colchetes), até que seja decidido se a função está no caso recursivo ou não. Quando isto é constatado, após a instrução `JFALSE`, uma nova aplicação `$1 (3)` é contruida, colocada na pilha e avaliada, pela aplicação dos mesmos passos descritos acima, até que o caso resursivo não possa ser aplicado.

3.8.1 O Interpretador

Na implementação de um interpretador para GCODE, que implemente a funcionalidade de uma máquina G, são necessários os seguintes componentes adicionais:

- **PC**: contador de programa, aponta para uma instrução em **C** que está sendo executada;
- **T**: tabela de associação, contendo pares de valores, onde o primeiro componente é um nome e o segundo é um apontador para uma instrução em **C**;

GCODE	Stack	Dump
	$1 [(1,4)]	(,EVAL)
UNWIND	[(1,4)] $1	(,EVAL)
PUSH 0	[(1,4)] [(1,4)] $1	(,EVAL)
GET	(1,4) [(1,4)] $1	(,EVAL)
PUSHVAL 1 0	[(1,0)] (1,4) [(1,4)] $1	(,EVAL)
GET	(1,0) (1,4) [(1,4)] $1	(,EVAL)
EQ	(3,0) [(1,4)] $1	(,EVAL)
NEG	(3,1) [(1,4)] $1	(,EVAL)
JFALSE L0	[(1,4)] $1	(,EVAL)
PUSH 0	[(1,4)] [(1,4)] $1	(,EVAL)
GET	(1,4) [(1,4)] $1	(,EVAL)

GCODE	Stack	Dump
PUSH 1	[(1,4)] [(1,4)] $1	(,EVAL)
GET	(1,4) [(1,4)] $1	(,EVAL)
PUSHVAL 1 1	[(1,1)] (1,4) [(1,4)] $1	(,EVAL)
GET	(1,1) (1,4) [(1,4)] $1	(,EVAL)
SUB	(1,3) [(1,4)] $1	(,EVAL)
MKVAL	[(1,3)] [(1,4)] $1	(,EVAL)
PUSHGLOBAL $1	$1 [(1,3)] [(1,4)] $1	(,EVAL)
MKAP 1	$1 [(1,3)] [(1,4)] $1	(,EVAL)
EVAL	$1 [(1,3)]	[(1,4)] ($1 ,EVAL) (,EVAL)

Figura 3.11: **Execução do caso recursivo em $1 (4) (= fat 4)**

A tabela **T** contém rótulos que aparecem em instruções `JUMP` ou `JFALSE`, e nomes de combinadores usados no programa. Associados a estes estão as instruções subsequentes à `LABEL` ou à `GLOBALSTART`, respectivamente. Já que estas últimas são pseudo-instruções, **T** não seria necessária se optassemos por gerar código equivalente às instruções GCODE. Neste caso, os nomes seriam substituidos pelos endereços físicos das instruções.

O *heap* é implementado na forma de uma lista de apontadores para as células de mémória alocadas. Cada uma destas células contabiliza o número de referências a ela, e quando uma nova célula é necessária, ocorre uma busca sobre o grafo, a procura daquelas células com número de referências zerado. Se existir alguma que tenha o mesmo tipo da célula que é demandada, um apontador para esta é apresentado como resultado. Caso contrário uma nova célula é alocada. Nesta busca, as células de outros tipos com zero referências são dealocadas. A estratégia de reutilizar células sem referência diminui a fragmentação de memória.

A interpretação de um programa é feita pela ativação de um método *Exec* pertencente ao objeto que representa a GMachine em tempo de execução. Este é implementado como um laço, que ativa a cada passo o método de execução da instrução apontada por **PC**, seguido do incremento deste apontador. As instruções do programa em **C** são representadas da forma apresentada anteriormente.

3.8.2 O Depurador

A construção de depuradores para interpretadores implementados de forma organizada é trivial, fato constatado inclusive por outros autores, consumindo apenas algumas horas de programação.

No nosso caso, o método *Exec* é substituido por *Debug*, que passa a controlar também a interface com o usuário, e habilita que este fique responsável pela decisão sobre a execução ou não das instruções do programa. Esta interface é apresentada na figura 3.12.

```
GMachine Debugger Version 1.0

command> h
Help:
clear    - reset machine state
dump     - print dump contents
exec     - execute the program
graph    - print heap usage
help     - show help information
load     - load a program
next     - execute next instruction
program  - list instructions
quit     - leave the debugger
stack    - print stack contents
trace    - show program instructions during its execution

command>
```

Figura 3.12: **Interface do Depurador**

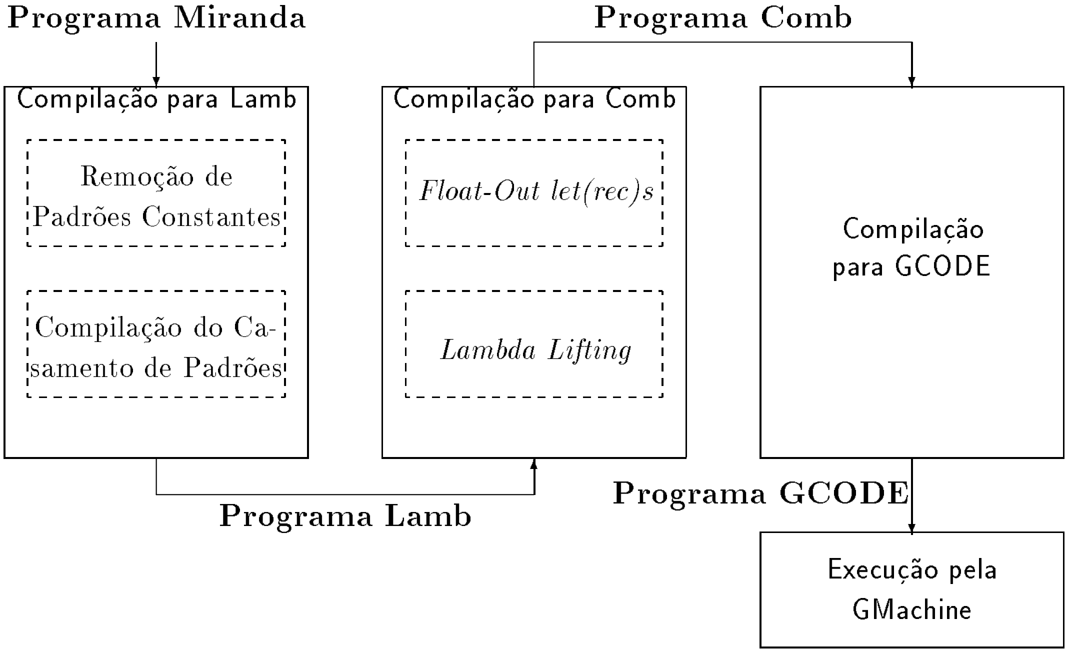

Figura 3.13: **Processo de compilação e interpretação**

3.9 Sumário

Na figura 3.13 é apresentado um diagrama que sumariza os principais passos dos processos de compilação e interpretação discutidos neste capítulo. Na figura, não aparece representada a Análise de Dependências, pois esta ocorre dispersa pelos vários sub-processos, e sob demanda.

Capítulo 4

Considerações Finais

"A great discovery solves a great problem but there is a grain of discovery in the solution of any problem".

G. Polya

Neste trabalho, apresentamos um compilador/interpretador para a linguagem funcional Miranda, desenvolvido usando um método que conjuga o uso de orientação a objetos como enfoque para o processo de desenvolvimento ao uso de especificações formais de linguagens de programação como técnica de desenho.

O método proposto utiliza as informações presentes na especificação da gramática de cada linguagem para derivar estruturas de classes organizadas sob a forma de herança e composição. Usando esquemas de tradução são derivados métodos para cada uma dessas classes, explorando o polimorfismo presente em ambas as construções.

Nos casos em que foi aplicado, o método se mostrou bastante satisfatório, organizando e sistematizando o processo de desenvolvimento, separando sintática e semânticamente as linguagens envolvidas no processo de compilação, e permitindo que os diferentes tipos de componentes necessários a um compilador pudessem ser desenhados de forma diferenciada:

- componentes dependentes da linguagem de programação: foram derivados usando um conjunto de regras precisas.

- componentes independentes da linguagem de programação: foram criados dentro de um *framework* e reutilizados em cada caso.

A implementação do compilador, realizada ao longo de um ano de trabalho, totaliza atualmente 16000 linhas de código, escritas usando a linguagem C++ e as linguagens de especificação das ferramentas LEX e do YACC. Este produto é portátil, podendo ser executado sobre os sistemas operacionais MS-DOS[1] e UNIX[2].

Sobre o uso da linguagem C++, pudemos concluir que esta apresenta uma série de inconvenientes no desenvolvimento de projetos do porte apresentado neste trabalho. Em primeiro lugar, quando se explora todo o potencial da linguagem, com o uso de herança múltipla e classes virtuais, é necessário saber como os programas que se estruturam dessa forma são compilados, para que se possa trabalhar com os respectivos apontadores para objetos. Além disso, características como a coleta de lixo, a identificação única de objetos e suas classes, e o tratamento de objetos persistentes tem de ser implementados, não sendo suportados diretamente. Apesar desses problemas, não é possível encontrar atualmente outra linguagem que a substitua sem que seja perdida a característica de portabilidade da implementação.

Foram integradas no compilador várias técnicas para compilação de programas funcionais, como a análise de dependências, o casamento de padrões e o *fully lazy lambda-lifting*. Em alguns casos se alcançou melhorias e otimizações na compilação (realização de menos buscas sobre a árvore que representa o programa durante a análise de dependências) e no código gerado (é usada apenas uma instrução para tratar valores básicos, e não uma para cada tipo), comparando esta a outras propostas.

Em linhas gerais, pudemos notar que o uso de orientação a objetos permitiu que a implementação se tornasse bastante modular, apresentando toda a clareza conceitual que pode ser percebida nas especificações formais. Estas últimas serviram inclusive como documentação do produto, ajudando na correção de alguns erros de implementação. Com isto, o processo de desenvolvimento e a implementação final se tornaram bastante organizados, confirmando nossa tese de que o uso conjunto deste enfoque e desta técnica é benéfico. Com estas características, foram criados produtos extensíveis, como comentamos adiante.

[1]MS-DOS é uma marca registrada da Microsoft Corporation

[2]UNIX é uma marca registrada da AT&T Bell Laboratories

4.1 Outros trabalhos relacionados

O uso de orientação a objetos no processo de contrução de compiladores é proposto por Crensaw em [Cre91]. Entretanto, este autor não mostra claramente como representar as relações entre os símbolos da linguagem que aparecem especificados nas produções do BNF. Também não é abordado naquele trabalho como tratar da semântica de linguagens de programação.

Por outro lado, embora seja freqüente encontrar referências que defendam que o uso de especificações formais de linguagens de programação (principalmente da semântica destas) é útil na implementação de compiladores, como em [Pag81], [All86], [Sch88] e [GM87], não se conhece uma proposta que faça uso e se beneficie de um enfoque específico para o processo de desenvolvimento como apresentado neste trabalho.

Buscando estruturar melhor suas implementações e preservar nestas as características formais da expecificação, autores como Rothwell [Rot92], Meira [MMS91] e Peyton-Jones [Jon87, JL92, Jon92] utilizam múltiplas linguagens na compilação de programas funcionais, e principalmente uma semelhante ao *lambda calculus*. Parece ser um idéia unânime entre eles que modularidade e extensibilidade são alcançadas dessa forma.

Além disso, "... modularidade e abstração são inicialmente mais importantes que eficiência crua - um compilador bem estruturado pode ser feito mais eficiente otimizando os módulos e refinando as interfaces entre eles, mas um compilador eficiente raramente pode ser incrementado em modularidade e estrutura", segundo Rothwell.

Quanto à implementação de compiladores para linguagens funcionais de programação, percebe-se que os vários modelos de redução e de máquinas abstratas levam a arquiteturas de software que tem entre si mais semelhanças que diferenças. Por exemplo, o algorítmo escolhido neste trabalho para realização de análises de dependências é idêntico ao usado por Peyton-Jones [JL92], sem que tivessemos conhecimento anterior sobre tal trabalho. Constata-se que a coincidência entre estes trabalhos é ainda maior observando que ambas as implementações do "lambda-lifting" são extremamente parecidas.

A arquitetura de cada compilador criado como sugerido neste trabalho é bastante semelhante àqueles criados usando a meta-ferramenta DRACO descrita por Neighbours em

[Nei84]. Enquanto naquele trabalho a ênfase maior é sobre a reutilização das implementações e do desenho de cada componente das linguagens, em nosso trabalho enfatizamos o processo de tradução e representação destas, usando especificações formais como ponto de partida.

4.2 Trabalhos Futuros

Além de necessitarem ser melhor testados e experimentados, os produtos deste trabalho (o método e o compilador) podem ser extendidos de várias formas. Algumas destas extensões já se encontram inclusive em fase de implementação.

4.2.1 Extensões ao compilador e às linguagens

Sem que se perca a organização alcançada neste trabalho, podem ser feitas as seguintes extensões ao compilador e às respectivas linguagens:

1. *Type Checking*: Linguagens funcionais de programação se utilizam normalmente de sistemas de tipos recursivos e polimórficos. Nas melhores implementações, a checagem de tipos é feita por um algorítmo de inferência baseado em unificação, que dá liberdade ao programador para declarar ou não os tipos das declarações [Jon87]. A partir do programa é feita a determinação dos tipos de todas as declarações, onde estes não aparecem explícitos. Esta atividade é realizada em tempo de compilação (*static type-checking*), e é ortogonal ao processo apresentado neste trabalho.

   Para integrar a checagem de tipos ao nosso compilador, seria necessário apenas declarar mais um componente na classe *ProgramSymbol*, para apontar o tipo de cada símbolo. Esquemas especiais seriam usados para especificar e derivar o componente de software que implementa esta função. Esta atividade seria realizada antes da tradução de programas escritos em qualquer das linguagens utilizadas. Na implementação, um algorítmo capaz de tratar tipos recursivos e polimórficos deveria ser usado, como o proposto por Palsberg [KPS93], com complexidade $O(n^2)$.

2. *Strictness Analysis*: É outra análise que pode ser realizada sobre programas funcionais. Esta envolve a criação de uma interpretação abstrata do programa, que permite a iden-

tificação de funções e expressões que realmente terminam, a um custo bem menor que se aplicada ao programa original. Estes casos são traduzidos em um código otimizado. "Usando esta tecnologia, compiladores para linguagens preguiçosas podem gerar código que é em alguns casos tão ou mais rápido que C", segundo Peyton-Jones [Jon92].

Para integrar esta análise ao nosso compilador, deve ser criada uma nova linguagem, usada para representar programas abstratos, derivados daqueles escritos na linguagem Comb. As classes que representam símbolos desta linguagem devem ter acrescentado à suas estruturas um componente dizendo se este é *strict* ou não. A correspondência entre as linguagens e as regras de verificação seriam especificados através de esquemas especiais.

3. *Error Handling*: Como nos programas gerados na linguagem GCODE existe a possibilidade de se encaixar código para tratar da recuperação de erros, é possível que as outras linguagens usadas tenham esta característica. Assim, basta escolher uma forma para que isto possa ser feito sem quebrar o estilo funcional de programação. Idéias como as apresentadas por Govindarajan [Gov93] ou por Rothwell [Rot92] podem ser usadas para este fim.

4.2.2 Extensões ao interpretador

A estrutura formal da máquina G não possui componente que permita tratar da entrada de dados. Para que seja possível construir programas interativos, é necessário portanto acrescentar um componente **I** à máquina, e compilar os programas fonte para GCODE acrescida de uma instrução similar a `PRINT` para tratar a entrada de dados.

Outra extensão, proposta por autores como Palsberg [KPS93] e Peyton-Jones [Jon92], é a eliminação dos *tags* que informam o tipo de cada célula do *heap*. Esta pressupõe que cada instrução, ao ser executada, irá tomar valores corretos como entrada, não sendo necessário checar qual o tipo destes para que seja alcançado o comportamento desejado, economizando tempo e memória. Esta otimização só é possível quando *type checking* é realizado previamente e quando as instruções da linguagem objeto (GCODE no nosso caso) não dependem do tipo dos dados que manipulam.

4.2.3 Extensões ao depurador

Neste trabalho foi apresentado um depurador para a linguagem GCODE. Para que seja possível depurar também programas fonte, é necessário acrescentar entre os componentes de *ProgramSymbol* um apontador para seu significado, dado na linguagem objeto. Assim, em cada passo da interação com o usuário, o depurador executaria todas as instruções GCODE associadas à instrução corrente do programa fonte.

4.2.4 Extensões ao método

A medida que o método proposto neste trabalho foi evoluindo, e seu nível de formalização aumentando, percebeu-se que um gerador de compiladores poderia ser criado, semelhante ao proposto em [Pal92]. Esta ferramenta estaria baseada na especificação formal de linguagens de programação, sendo criado usando o *framework* e as regras descritas no capítulo 2. Neste gerador, poderiam ser usadas as seguintes linguagens para especificação da semântica:

- linguagem de definição de esquemas de tradução: a especificação da linguagem fonte seria usada para definir como programas seriam traduzidos em uma linguagem objeto já especificada.
- linguagem denotacional: a especificação da linguagem fonte seria usada para definir como programas escritos em Lamb seriam executados.

O estilo transformacional usado neste trabalho é amplamente usado na especificação de linguagens funcionais de programação. Entretanto isto não ocorre com outros tipos de linguagem. Portanto, a segunda opção parece oferecer mais atrativos, por definir o compilador de forma mais abstrata e padronizada.

Bibliografia

[All86] Loyd Allison. *A practical introduction to denotational semantics.* Cambridge University Press, 1986.

[BC79] William A. Barret and John D. Couch. *Compiler Construction: Theory and Practice.* Ed. Science Research Associates Inc, 1979.

[Cre91] Jack W. Crenshaw. A perfect marriage. *Computer Language*, 8(6):44–55, June 1991.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[Dow86] M. Dowson. The Structure of the Software Process. *ACM SIGSOFT Software Engineering Notes*, 11(4):6–8, August 1986.

[GM87] Joseph A. Goguen and Mark Moriconi. Formalization in programming environments. *IEEE Computer*, pages 55–64, November 1987.

[Gov93] R. Govindorajan. Exception Handlers in Functional Programming Languages. *IEEE Transaction on Software Engineering*, 19(8):826–834, August 1993.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[JL92] Simon L. Peyton Jones and David Lester. *Implementating Functional Languages: A Tutorial.* Ed. Prentice Hall, 1992.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Ed. Prentice Hall, 1987.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless-Tagless G-machine: Version 2.5. Department of Computing Science, University of Glasgow, July 1992.

[JW91] Simon L. Peyton Jones and Philip Wadler. A static semantics for HASKELL. Department of Computing Science, University of Glasgow, May 1991.

[KPS93] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *ACM Symposium on Principles of Programming Languages (POPL)*, January 1993.

[MMS91] Silvio Lemos Meira, Martin A. Musicante, and André Santos. O projeto e a implementação da linguagem A. In *Anais do V Simpósio Brasileiro de Engenharia de Software*, pages 237–256, October 1991.

[Nei84] J. M. Neighbors. The draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, 10(5):564–574, 1984.

[Pag81] Frank G. Pagan. *Fomal Specification of Programming Languages: A Panoramic Primer*. Ed. Prentice Hall, 1981.

[Pal92] Jens Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Aarhus University, 1992.

[R+91] James Rhumbauh et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Rot92] Nick Rothwell. Functional compilation from the standard ML core language to lambda calculus. Technical Report ECS-LFCS-92-235, Department of Computer Science, University of Edimburg, September 1992.

[Sch88] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. C. Brown Publishers, 1988.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Tur86] David Turner. An overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, December 1986.

[WBJ90] Rebecca J. Wirfs-Brock and Ralph E. Johnson. Surveying current research in object-orientated design. *Communications of the ACM*, 33(9):104–124, September 1990.

Apêndice A

Sintaxe de Miranda

$$
\begin{array}{lll}
statement\_list & \rightarrow & \epsilon \\
 & | & statement\ statement\_list \\
statement & \rightarrow & \text{IDENTIFIER}\ pattern\_list\ guard\_list \\
 & | & \text{IDENTIFIER}\ pattern\_list\ guard\_list\ \textbf{where}\ statement\_list\ \text{SEMI} \\
guard\_list & \rightarrow & guard\ |\ normal\_guard\ guard\_list \\
guard & \rightarrow & normal\_guard\ |\ otherwise\_guard \\
normal\_guard & \rightarrow & \text{EQ}\ expression\ \text{COMMA}\ expression \\
otherwise\_guard & \rightarrow & \text{EQ}\ expression \\
 & | & \text{EQ}\ expression\ \text{COMMA}\ \textbf{otherwise} \\
pattern\_list & \rightarrow & \epsilon \\
 & | & pattern\ pattern\_list \\
pattern & \rightarrow & ident\_pattern \\
 & | & num\_pattern \\
 & | & val\_pattern \\
 & | & char\_pattern \\
 & | & list\_pattern \\
 & | & op\_pattern \\
 & | & tuple\_pattern \\
 & | & struct\_pattern \\
 & | & \text{LP}\ pattern\ \text{RP}
\end{array}
$$

$ident\_pattern \rightarrow$ IDENTIFIER

$num\_pattern \rightarrow$ NUMBER

$val\_pattern \rightarrow$ VALUE

$char\_pattern \rightarrow$ CHAR

$list\_pattern \rightarrow list$

$op\_pattern \rightarrow$ LP $operator$ RP

$tuple\_pattern \rightarrow$ LP $tuple\_list$ RP

$struct\_pattern \rightarrow$ LP VALUE $parameter\_list$ RP

$unary\_operator \rightarrow$ SUCC | PRED | NOT | CARD

$binary\_left\_add\_oper \rightarrow$ CONC | REM | PLUS | MINUS

$binary\_left\_mult\_oper \rightarrow$ DIV | MULTI

$binary\_left\_cmp\_oper \rightarrow$ IN

$binary\_right\_cmp\_oper \rightarrow$ L_AND | L_OR | EQUAL | NEQ
| LE | LEQ | GT | GEQ

$binary\_left\_other\_oper \rightarrow$ SUBSCRIPT

$binary\_operator \rightarrow binary\_right\_cmp\_oper$
| $binary\_left\_cmp\_oper$
| $binary\_left\_other\_oper$
| $binary\_left\_mult\_oper$
| $binary\_left\_add\_oper$

$operator \rightarrow unary\_operator \mid binary\_operator$

$expression \rightarrow error\_expression$
| $pattern\_expression$
| $operation\_expression$

$operation\_expression \rightarrow operation$

$error\_expression \rightarrow$ **error** STRING

$pattern\_expression \rightarrow pattern$

$operation \rightarrow unary\_operation$
| $operator\_binary\_operation$
| $identifier\_binary\_operation$
| $funct\_call\_operation$
| LP $operation$ RP

operator_binary_operation → expression binary_right_cmp_oper expression
| expression binary_left_cmp_oper expression
| expression binary_left_other_oper expression
| expression binary_left_mult_oper expression
| expression binary_left_add_oper expression

identifier_binary_operation → expression DOLLAR IDENTIFIER expression

unary_operation → unary_operator expression
| LP MINUS expression RP

funct_call_operation → IDENTIFIER LP parameter_list RP

list → string
| head_tail_list
| empty_list
| enumerated_list
| list_comprehension

string → STRING

head_tail_list → expression CONS expression

empty_list → EMPTY_LIST

enumerated_list → LB contents_list RB

list_comprehension → LB contents_list VBAR qualifier_list RB

contents_list → list_contents
| list_contents COMMA contents_list

list_contents → single_list_element | to_list_element
| from_list_element | from_to_list_element

from_to_list_element → expression DOTDOT expression

single_list_element → expression

from_list_element → expression DOTDOT

to_list_element → DOTDOT expression

qualifier_list → expression
| expression SEMI qualifier_list

parameter_list → expression
| expression parameter_list

tuple_list → expression COMMA expression
| expression COMMA tuple_list

Apêndice B

Sintaxe de Lamb

$$
\begin{aligned}
lambda\_spec &\rightarrow \epsilon \mid lambda\_expr \\
lambda\_expr &\rightarrow node \\
&\mid aplication \\
&\mid abstraction \\
&\mid let\_expr \\
&\mid letrec\_expr \\
&\mid \mathrm{LP}\ lambda\_expr\ \mathrm{RP} \\
node &\rightarrow var\_node \mid val\_node \mid code\_node \\
var\_node &\rightarrow \mathrm{IDENTIFIER} \\
code\_node &\rightarrow \mathrm{VALUE} \\
val\_node &\rightarrow \mathrm{FALSE} \mid \mathrm{TRUE} \mid \mathrm{NIL} \mid \mathrm{CONS} \mid \mathrm{NUMBER} \mid \mathrm{CHAR} \\
aplication &\rightarrow \mathrm{LP}\ lambda\_expr\ lambda\_expr\ \mathrm{RP} \\
abstraction &\rightarrow \mathrm{LAMBDA}\ var\_node\ \mathrm{DOT}\ lambda\_expr \\
constant\_def &\rightarrow var\_node\ \mathrm{EQ}\ lambda\_expr \\
let\_expr &\rightarrow \textbf{let}\ constant\_def\ \textbf{in}\ lambda\_expr \\
letrec\_expr &\rightarrow \textbf{letrec}\ lambda\_expr\_list\ \textbf{in}\ lambda\_expr \\
lambda\_expr\_list &\rightarrow \epsilon \mid constant\_def\ \mathrm{COMMA}\ lambda\_expr\_list
\end{aligned}
$$

Apêndice C

Sintaxe de Comb

$$
\begin{array}{lcl}
\mathit{combinator\_spec} & \rightarrow & \epsilon \mid \mathit{combinator\_list} \\
\mathit{combinator\_list} & \rightarrow & \mathit{combinator\_def} \mid \mathit{combinator\_def}\ \mathit{combinator\_list} \\
\mathit{lambda\_expr} & \rightarrow & \mathit{node} \\
 & \mid & \mathit{aplication} \\
 & \mid & \mathit{let\_expr} \\
 & \mid & \mathit{letrec\_expr} \\
 & \mid & \mathrm{LP}\ \mathit{lambda\_expr}\ \mathrm{RP} \\
\mathit{node} & \rightarrow & \mathit{var\_node} \mid \mathit{val\_node} \mid \mathit{code\_node} \\
\mathit{var\_node} & \rightarrow & \mathrm{IDENTIFIER} \\
\mathit{code\_node} & \rightarrow & \mathrm{VALUE} \\
\mathit{val\_node} & \rightarrow & \mathrm{FALSE} \mid \mathrm{TRUE} \mid \mathrm{NIL} \mid \mathrm{CONS} \mid \mathrm{NUMBER} \mid \mathrm{CHAR} \\
\mathit{aplication} & \rightarrow & \mathrm{LP}\ \mathit{lambda\_expr}\ \mathit{lambda\_expr}\ \mathrm{RP} \\
\mathit{constant\_def} & \rightarrow & \mathit{var\_node}\ \mathrm{EQ}\ \mathit{lambda\_expr} \\
\mathit{let\_expr} & \rightarrow & \mathbf{let}\ \mathit{constant\_def}\ \mathbf{in}\ \mathit{lambda\_expr} \\
\mathit{letrec\_expr} & \rightarrow & \mathbf{letrec}\ \mathit{lambda\_expr\_list}\ \mathbf{in}\ \mathit{lambda\_expr} \\
\mathit{lambda\_expr\_list} & \rightarrow & \epsilon \mid \mathit{constant\_def}\ \mathrm{COMMA}\ \mathit{lambda\_expr\_list} \\
\mathit{var\_expr\_list} & \rightarrow & \epsilon \mid \mathit{var\_node}\ \mathrm{COMMA}\ \mathit{var\_expr\_list} \\
\mathit{combinator\_def} & \rightarrow & \mathit{var\_node}\ \mathrm{LP}\ \mathit{var\_expr\_list}\ \mathrm{RP}\ \mathrm{EQ}\ \mathit{lambda\_expr}
\end{array}
$$

Apêndice D

Sintaxe de Gcode

$$
\begin{array}{lcl}
\mathit{g\_spec} & \rightarrow & \epsilon \mid \mathit{gprogram} \\
\mathit{gprogram} & \rightarrow & \mathit{ginstruction} \\
 & \mid & \mathit{ginstruction}\ \mathrm{SEMI}\ \mathit{gprogram} \\
\mathit{ginstruction} & \rightarrow & \mathit{add} \mid \mathit{alloc} \mid \mathit{begin} \mid \mathit{div} \mid \mathit{end} \mid \mathit{eq} \mid \mathit{eval} \mid \mathit{ge} \mid \mathit{get} \mid \mathit{gt} \\
 & \mid & \mathit{global\_start} \mid \mathit{jfalse} \mid \mathit{jump} \mid \mathit{label} \mid \mathit{le} \mid \mathit{lt} \mid \mathit{mkap} \mid \mathit{mkval} \mid \mathit{mul} \\
 & \mid & \mathit{neg} \mid \mathit{neq} \mid \mathit{pop} \mid \mathit{print} \mid \mathit{push} \mid \mathit{push\_global} \mid \mathit{push\_val} \mid \mathit{return} \\
 & \mid & \mathit{slide} \mid \mathit{sub} \mid \mathit{unwind} \mid \mathit{update} \mid \mathit{update\_val}
\end{array}
$$

$\mathit{add} \rightarrow \mathbf{add}$

$\mathit{alloc} \rightarrow \mathbf{alloc}\ \mathrm{NUMBER}$

$\mathit{begin} \rightarrow \mathbf{begin}$

$\mathit{div} \rightarrow \mathbf{div}$

$\mathit{end} \rightarrow \mathbf{end}$

$\mathit{eq} \rightarrow \mathbf{eq}$

$\mathit{eval} \rightarrow \mathbf{eval}$

$\mathit{ge} \rightarrow \mathbf{ge}$

$\mathit{get} \rightarrow \mathbf{get}$

$\mathit{gt} \rightarrow \mathbf{gt}$

$\mathit{global\_start} \rightarrow \mathbf{globalstart}\ \mathrm{VALUE}\ \mathrm{NUMBER}$

$\mathit{jfalse} \rightarrow \mathbf{jfalse}\ \mathrm{VALUE}$

$\mathit{jump} \rightarrow \mathbf{jump}\ \mathrm{VALUE}$

$label \rightarrow$ **label** VALUE

$le \rightarrow$ **le**

$lt \rightarrow$ **lt**

$mkap \rightarrow$ **mkap** NUMBER

$mkval \rightarrow$ **mkval**

$mul \rightarrow$ **mul**

$neg \rightarrow$ **neg**

$neq \rightarrow$ **neq**

$pop \rightarrow$ **pop** NUMBER

$print \rightarrow$ **print**

$push \rightarrow$ **push** NUMBER

$push\_global \rightarrow$ **pushglobal** VALUE

$push\_val \rightarrow$ **pushval** NUMBER NUMBER

$return \rightarrow$ **return**

$slide \rightarrow$ **slide** NUMBER

$sub \rightarrow$ **sub**

$unwind \rightarrow$ **unwind**

$update \rightarrow$ **update** NUMBER

$update\_val \rightarrow$ **updtval** NUMBER