Em Anais PANEL'95 - XXI Conferência Latino-Americana em Informática, 743–754. Julho/Agosto, 1995.

Aplicando um método orientado a objetos para desenvolvimento de compiladores*

Carlos H. C. Duarte

Department of Computing
Imperial College, London
cd7@doc.ic.ac.uk

Roberto Ierusalimschy

Departamento de Informática
PUC, Rio de Janeiro
roberto@inf.puc-rio.br

Resumo

A construção de compiladores é uma atividade complexa, que demanda a criação de especificações rigorosas de linguagens de programação para permitir que produtos sejam desenvolvidos e mantidos de maneira viável. Neste trabalho, um método sistemático para desenvolvimento de compiladores é proposto, conjugando orientação a objetos à especificação formal de linguagens de programação. A aplicação deste método no desenvolvimento de um compilador para uma linguagem funcional é descrita, mostrando que produtos modulares e estensíveis podem ser obtidos como conseqüência de um processo de desenvolvimento organizado para este domínio.

Abstract

Compiler construction is a complex activity, which demands the creation of rigorous programming language specifications to make possible the development and maintenance of products in a feasible manner. In this paper, a systematic method for compiler development is proposed, combining object orientation and formal specification of programming languages. The development of a functional language compiler using this method is described, showing that modular and extensible products can be obtained as the result of a development process organised to this domain.

*Este trabalho foi parcialmente financiado pelo CNPq: Conselho Nacional de Desenvolvimento Científico e Tecnológico.

1 Introdução

A construção de compiladores é uma atividade que tem apresentado problemas de difícil tratamento desde os primórdios da Ciência da Computação. A alta complexidade e a crescente dimensão inerentes à este tipo de software são fatores que tem contribuído para torná-los difíceis de desenvolver e manter.

Surpreendentemente, não são muitas as soluções propostas visando tratar destes fatores. Em geral, apenas os aspectos relacionados à corretude da implementação são considerados relevantes. Conforme notado por Goguen [4], embora formalismos se mostrem essenciais neste sentido, especificações incorretas e inadequações notacionais têm freqüentemente levado a experiências mal sucedidas. Por outro lado, algumas propostas simplesmente não tratam do grande número de características dependentes das linguagens de programação presentes neste tipo de programa.

Recentemente, enfoques para o processo de desenvolvimento de software tem sido adotados buscando apresentar maneiras organizadas de contornar estes problemas, levando em conta que "... modularidade e abstração são inicialmente mais importantes que eficiência crua", segundo Rothweel [11]. Tendo em mente este fato, o enfoque orientado a objetos parece ser uma boa alternativa em direção a um processo organizado, tal como proposto inicialmente por Koskimies [9] e recentemente por Crenshaw [2]. Nestas propostas, embora os requisitos relacionados à corretude não sejam abordados, é provido suporte para a criação compiladores assumindo que as informações dependentes do domínio (linguagens de programação) podem ser representadas de forma adequada através de estruturas como classes e métodos.

Neste trabalho, um método sistemático para construção de compiladores é proposto, conciliando especificação formal de linguagens de programação ao uso de conceitos provenientes do enfoque orientado a objetos. Especificamente, a representação de linguagens e programas neste tipo de software e a tradução destes são tratados de forma destacada, visando alcançar características como modularidade e estensibilidade neste produtos, por serem consideradas primordiais no ciclo de vida de qualquer programa.

O método proposto foi validado no desenvolvimento de um compilador para uma linguagem funcional de alto nível, baseado nas idéias apresentadas inicialmente por Peyton-Jones [5]. Neste caso, toda a complexidade inerente à família das linguagens funcionais e a maioria dos problemas relacionados foram encontrados e tratados satisfatoriamente, sendo apresentados neste artigo como exemplo.

O restante do artigo está organizado da seguinte maneira: A segunda seção descreve o método proposto; em seguida são apresentados alguns aspectos do compilador desenvolvido; na última seção são feitas as considerações finais do trabalho.

2 O método para desenvolvimento de compiladores

O método proposto se apóia em especificações formais da linguagem de programação no desenho e na implementação do compilador. As especificações da sintaxe da linguagem são usadas para determinar quais as estruturas de dados adequadas, enquanto que a funcionalidade do compilador é derivada a partir da semântica formal da linguagem. Regras são fornecidas a seguir para guiar esta atividade de maneira sistemática.

2.1 Sintaxe

A especificação da gramática da linguagem de programação é usada em primeiro lugar na construção de um analisador léxico e sintático, e subseqüentemente na definição de estruturas para representar internamente os programas analisados. Considera-se neste trabalho que esta gramática é especificada sem ambigüidades por um conjunto de regras de produção na forma BNF, permitindo que geradores como *lex* e *yacc* [1] sejam usados para criar de forma automática o analisador apropriado.

Analisadores léxicos e sintáticos utilizam estruturas de dados para representar os programas analisados. Em [2] é apresentada uma regra básica para derivar tais estruturas usando orientação a objetos: "Para identificar os objetos requeridos por um tradutor, olhe para as equações sintáticas. Essencialmente, todo não-terminal representa um objeto". Como pré-requisito para aplicação desta regra, é necessário considerar que todos os elementos da sintaxe abstrata da linguagem são não-terminais, e somente estes.

Considere como exemplo as seguintes produções, que descrevem como listas podem ser definidas em uma linguagem funcional:

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

Usando-se a regra apresentada acima, *list*, *string*, *head_tail_list*, *empty_list*, *enumerated_list* e *list_comprehension* são representados durante a execução por objetos pertencentes às classes *List*, *String*, *HeadTailList*, *EmptyList* e *ListComprehension*, respectivamente.

Além de nomes de símbolos, existem várias outras informações presentes na especificação de qualquer gramática. Estas se referem à organização dos símbolos, e assim devem apontar a estrutura das classes que os representam.

A definição de várias produções tendo em comum o mesmo símbolo do lado esquerdo indica que este pode ser usado em um contexto mais genérico para substituir os símbolos que aparecem do lado direito de cada regra. Esta característica pode ser facilmente modelada através do conceito de herança. Neste caso, as classes que representam os símbolos à direita devem ser sub-classes daquela que representa o símbolo à esquerda nas produções. No exemplo anterior, *String*, *HeadTailList*, *EmptyList*, *EnumeratedList* e *ListComprehension* devem ser sub-classes de *List*.

É de se esperar que as produções onde símbolos são definidos por agregação dêem origem a uma classe composta pela agregação das classes que representam os símbolos constituintes. Do seguinte exemplo[1],

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

[1] Símbolos terminais são representados em maiúsculas.

a classe *HeadTailList* deve ser criada compreendendo duas ocorrências de *Expression*, ocupando respectivamente os papéis de cabeça e cauda da lista.

Na tabela 1 são sumarizadas as regras apresentadas nesta seção. Adicionalmente, outras regras podem ser utilizadas, buscando obter uma hierarquia de classes mais adequada, através da generalização de componentes comuns, por exemplo. Maiores detalhes a este respeito podem ser encontrados em [3].

Na gramática	**Na representação**
terminal	atributo
não-terminal	classe abstrata
produção	classes concretas
{ várias produções para um símbolo	{ sub-classe
{ vários não-terminais em uma produção	{ agregação

Tabela 1: Regras para derivação de classes a partir da gramática da linguagem

2.2 Semântica

Programas entregues à entrada de um compilador são traduzidos sucessivamente até que uma representação "executável" seja produzida. Através da aplicação das regras apresentadas anteriormente para representar também os símbolos da linguagem objeto, cada etapa do processo de tradução pode ser decomposta em métodos pertencentes às classes da linguagem fonte, apresentando como resultado objetos das classes que representam a linguagem objeto. Esta representação permite que o polimorfismo suportado pelo enfoque orientado a objetos adotado seja explorado.

O enfoque transformacional usado na especificação formal da semântica de linguagens de programação [5] se adequa perfeitamente às estruturas criadas pela aplicação das regras propostas. Esquemas de tradução são usados para descrever o significado de cada símbolo em função de outros com significado conhecido (normalmente pertencente à outra linguagem). A notação utilizada aqui é semelhante à apresentada em [5, 7], onde cada esquema possui uma assinatura e uma definição na seguinte forma:

nome_esquema : *símbolo_domínio* $\rightarrow$ *símbolo_imagem*
nome_esquema$[\![$*padrão_origem*$]\!]$ *(parâmetros)* $\triangleq$ *padrão_destino*

Padrões são compostos por constantes, variáveis ou símbolos e devem satisfazer à sintaxe do domínio e da imagem de cada esquema. Parâmetros e nomes de esquemas de tradução podem ser usados para definir o *padrão_destino*. Esta representação do processo de tradução é bastante propícia, pois uma das características principais deste tipo de esquema é o seu polimorfismo, e o mesmo ocorre com métodos de classes organizadas por herança.

Exemplificando, considere a produção e o esquema apresentados na Figura 1, que estabelecem a sintaxe e a semântica de listas definidas por seus elementos. O

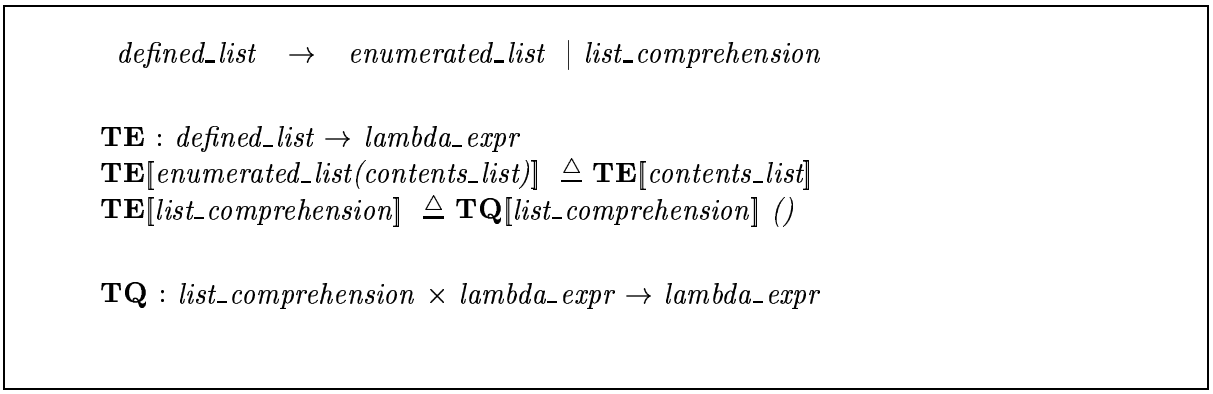

Figura 1: Sintaxe e Semântica de listas definidas por elementos

significado do respectivo símbolo é dado por **TE**. Se este símbolo for uma enumeração, sua tradução corresponderá à tradução de seu componente *contents_list*; se este for uma lista definida pelas propriedades de seus elementos, então seu significado será dado pelo esquema **TQ**. Neste exemplo, a representação de **TE** é o método `TE`, nas classes *DefinedList*, *EnumeratedList* e *ListComprehension*, que apresenta objetos da classe *LambdaExpr* como resultado.

Existem casos em que um esquema de tradução faz uso de outras informações além dos componentes do símbolo sobre o qual está sendo aplicado. Isto é o que ocorre com **TQ**, na Figura 1, 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 requer a definição de um método que receba parâmetros: `TQ` é um método em *ListComprehension* recebendo objetos da classe *LambdaExpr*.

Na criação de esquemas de tradução, existem decisões que competem ao especificador. Por exemplo, se é necessária a especificação de um esquema que atue sobre um conjunto de símbolos que não estão relacionados diretamente pela gramática da linguagem, a representação deste esquema estará dissociada de qualquer hierarquia de classes, sendo representado por uma função e não por um método. Casos como estes são apresentados a seguir, na seção 3.1.

3 O compilador para uma linguagem funcional

Recentemente, linguagens funcionais polimórficas, preguiçosas (*lazy*) e de alta ordem tem despertado atenção sobre si e sobre seu processo de compilação. Pertencem à esta classe Miranda[2][13], Haskell [8], A [10] e outras. O desenvolvimento de um compilador para um *subset* de Miranda é apresentado nesta seção.

A organização do compilador é detalhada na Figura 2. Nesta pode-se observar que os programas manipulados durante o processo de tradução podem estar representados de duas formas: textual ou na forma de árvore. A segunda representação pode ser obtida a partir da primeira através da análise sintática do texto do programa, enquanto

[2]Miranda é uma marca registrada da Research Software Limited.

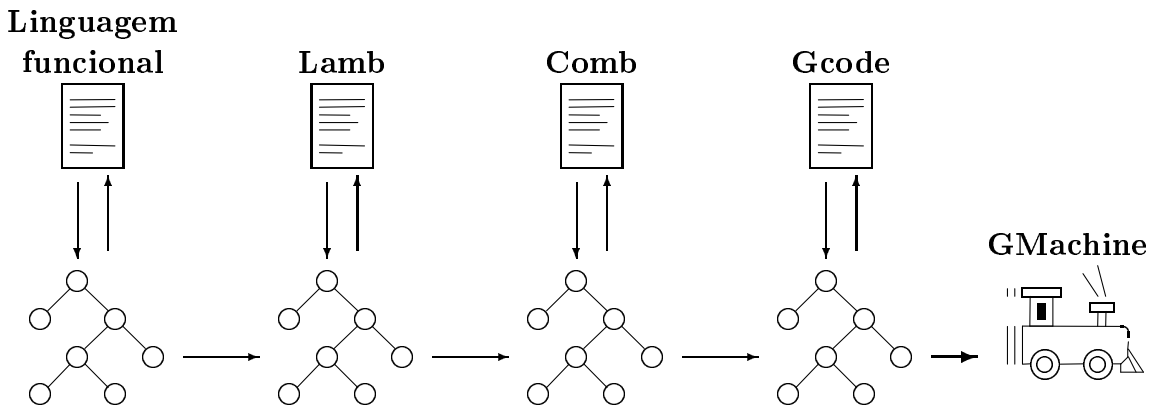

Figura 2: Visão organizacional do compilador

que a transformação oposta é feita pela impressão formatada (*pretty-print*) da árvore correspondente.

Cada programa apresentado como entrada ao compilador é traduzido sucessivamente em outros equivalentes a ele, escritos em linguagens de mais baixo nível. Cada uma destas linguagens é representada pela respectiva hierarquia de classes, sendo chamadas de **L.F.**, **Lamb**, **Comb** e **Gcode**, respectivamente. Ao final deste processo é obtido um programa que pode ser executado pelo interpretador apropriado, chamado **GMachine**.

3.1 Linguagens funcionais e sua compilação para Lamb

Um programa funcional é geralmente chamado *script*, e constituído por uma lista de declarações compostas por conjuntos de equações. Cada equação define parcialmente uma função, informando um sub-conjunto do domínio e definindo como é computada a imagem neste caso. Um *script* típico é apresentado na Figura 3.

As expressões usadas para definir o domínio de aplicação de cada função são chamadas padrões, e, quando existem, são compostas de forma análoga à descrita na seção 2.2. Na Figura 3, após o símbolo `=` em cada equação, aparecem termos "guardados", compostos por uma expressão simples e uma condição (chamada guarda). Por exemplo, `n * (n - 1)` tem `n > 0` como guarda. Quando um guarda é avaliado como verdadeiro, a respectiva equação é usada para computar o valor da função.

O esquema **TF** apresentado na Figura 4 define a semântica de listas de declarações (*statement_list*). Este método estabelece como um dado *script* é traduzido em

```
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: Exemplo de programa em uma linguagem funcional

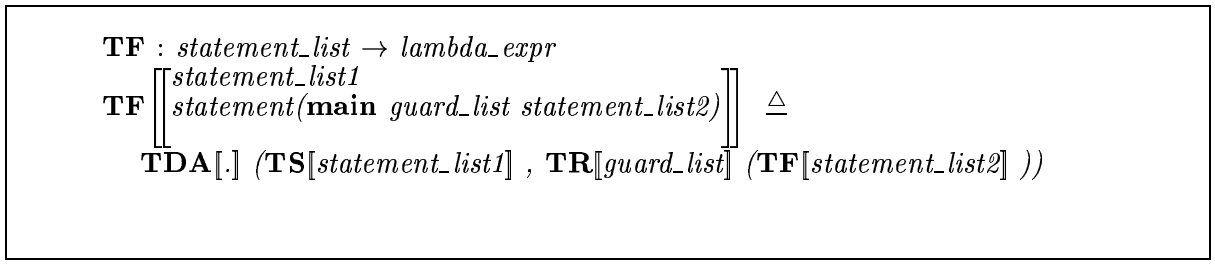

Figura 4: Esquema de tradução **TF**

uma árvore de objetos em **Lamb** pela aplicação de **TS**, **TF** e **TR**, sobre os componentes do programa. A representação deste esquema é o método `TF` na classe *StatementList*, raiz da hierarquia de classes usada para representar programas nesta linguagem.

Em alguns casos, funções se intercalam à ativação de métodos, como `TDA` no caso anterior. Esta é usada para realizar a análise das dependências entre os termos que irão compor uma definição com escopo delimitado. O resultado desta análise é uma expressão na qual os termos apresentados como entrada tem escopo somente sobre aquelas expressões onde são realmente utilizados.

A tradução de definições feitas por casos também é feita por uma função. Estas podem ser implementadas de duas formas distintas: 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 as seguintes implementações:

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

Já que ambas equações tem que possuir o mesmo significado e a primeira forma é necessária para que traduções subseqüentes sejam realizadas, padrões constantes são removidos de funções definidas por casos.

Uma terceira função chamada `Match` é usada na tradução de cada *script* para realizar a compilação do casamento de padrões. Esta cria uma expressão lambda para englobar todas as equações de uma função. Tal expressão é capaz de determinar qual equação deve ser usada no computo da imagem, com base no casamento entre os parâmetros de chamada e os parâmetros formais de uma das equações. Uma definição bastante detalhada do respectivo esquema é apresentada em [5].

3.2 Lamb e a compilação para Comb

Lamb é uma extensão do *lambda calculus*. Nesta linguagem, programas são compostos por variáveis, constantes e aplicações, e ainda por abstrações (funções) representadas pelo símbolo `^`. Declarações com escopo delimitado também podem ser feitas, usando `lets` para definições não recursivas, ou `letrecs`. O *script* anterior, traduzido para esta linguagem, é apresentado na Figura 5.

Tanto **Lamb** quanto as linguagens de mais baixo nível representam constantes através de pares, onde o primeiro componente indica o tipo e o segundo o valor propriamente dito. Na Figura 5, `(1, 4)` representa o inteiro 4. Construtures de tipo também

```
let norm = ^_2.let _5 = (SELECT (1, 2) (1, 2)) _2
               in let _4 = (SELECT (1, 2) (1, 1)) _2
                   in sqr / + sq _4 sq _5 (1, 2)
 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 5: Programa exemplo na linguagem **Lamb**

são representados desta forma, como `(0,2)` que denota o construtor de estruturas de dados que possuem aridade 2. Esta uniformidade evita um problema identificado em [6]: "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".

A tradução de programas em **Lamb** utiliza a técnica chamada *lambda lift* para remover as variáveis livres de cada expressão, permitindo que seja gerado um código eficiente posteriormente. Para cada abstração cria-se a definição de uma função, apresentando como parâmetros, além da variável abstraída, aquelas que aparecem livres na expressão que define o resultado da abstração. Nos locais onde é usada, esta abstração é substituída pela função resultante, aplicada às variáveis que se tornaram parâmetros. Usando desta técnica, a abstração `^_1.norm ((0, 2) _1 _1)` é traduzida para

```
$2 _1 = $3 ((0, 2) _1 _1)
```

O esquema **TLL** especifica o *lambda lifting* de uma maneira incrementada. Não só variáveis livres são removidas, mas também sub-expressões que não contenham variáveis amarradas à abstração. Desta forma economiza-se tempo e memória durante a execução, avaliando-se cada expressão o menor número de vezes possível, dando origem a um compilador totalmente preguiçoso, ou *fully lazy*. Ao ser aplicada esta técnica à abstração de `_2`, é criada uma função com três parâmetros, onde `(SELECT (1, 2) (1, 2))` e `(SELECT (1, 2) (1, 1))` também aparecem substituídas por variáveis .

Na avaliação de uma expressão, pode ocorrer que o corpo de definições contidas em um `let(rec)` tenham de ser recomputadas desnecessariamente, implicando na perda da característica totalmente preguiçosa. Para evitar que isto ocorra, a técnica conhecida como *float-out* é usada, retirando do corpo das definições as declarações que não são essenciais onde aparecem inclusas. Esta técnica é aplicada antes do *lambda lifting*, sendo especificada pelo esquema de tradução **TFO**. Enquanto apenas a estrutura interna de definições é tratada por **TDA**, no esquema **TFO** são usados como argumentos quaisquer expressões onde uma definição possa aparecer inclusa.

Tanto **TFO** quanto **TLL** têm uma definição bastante extensa e complexa, sendo definidas por casos, para cada um dos símbolos da linguagem **Lamb**. Entretanto,

```
$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 sqr / + sq _4 sq _5 (1, 2)
main = $1 $2 - (1, 4)
```

Figura 6: Programa exemplo na linguagem **Comb**

a complexidade inerente à estes esquemas é dividida e tratada usando-se o polimorfismo dos métodos que os representam. O método `TLL` em um objeto da classe *Aplication* só necessita se auto-aplicar sobre os dois componentes do objeto, não necessitando saber quem são estes. Detalhes a respeito de tais esquemas e sua implementação podem ser encontrados em [3].

3.3 Comb e GCODE

Comb é uma linguagem análoga à **Lamb**, na qual são permitidas declarações de supercombinadores. Segundo a definição apresentada em [5], "um supercombinador $S é uma expressão lambda na forma $\lambda x_1.\lambda x_2....\lambda x_n.E$ onde E não é uma abstração, tal que $S não tem variáveis livres, cada expressão em E é um supercombinador e $n \geq 0$".

Supercombinadores propiciam outra representação para funções, eliminando o uso explícito de abstrações da linguagem. O programa exemplo anterior aparece na Figura 6 traduzido para **Comb**, onde supercombinadores são representados como

```
$S x1 x2 ... xn = E
```

O uso de supercombinadores permite que programas funcionais sejam compilados, e um código eficiente seja gerado, baseado na redução dos grafos que representam as funções computadas pelo programa em tempo de execução. Estas reduções são realizadas por uma máquina de pilha chamada **GMachine**, que interpreta a linguagem correspondente chamada **GCODE**.

A compilação de cada supercombinador é especificada pelo esquema **F**, na Figura 7. Este define que um contexto para a execução do combinador será criado, associando cada nome de parâmetro à uma posição na pilha. A partir daí, outros esquemas são aplicados recursiva e polimorficamente. Ao final deste processo, a função `fat` apresentada anteriormente é traduzida em um programa com 43 instruções.

F : *combinator* $\rightarrow$ *g_program*
F$[\![$*code_node var_expr_list lambda_expr*$]\!]$ $\triangleq$
globalstart *code_node* **Length**[.] *(var_expr_list);*
R$[\![$*lambda_expr*$]\!]$ *(var_expr_list)*

Figura 7: Esquema de tradução **F**

3.4 A implementação do compilador

A implementação do compilador se torna bastante direta com a utilização do método proposto neste trabalho. Cada regra na especificação do analisador sintático — a ser entregue como entrada a um gerador compatível com *yacc* — deve seguir um dos padrões apresentados abaixo:

```
nao_terminal : componente_1 ... componente_n
               { $$ = new NaoTerminal ( $1, ..., $n ); }
               | ...
nao_terminal1 : nao_terminal2
                { $$ = $1; }
                | ...
```

Usando a linguagem de programação C++ [12], cada ação semântica do analisador deve ser definida como um `new` seguido pelo construtor da respectiva classe, como mostrado no primeiro caso acima. C++ também torna bastante trivial a implementação das classes derivadas usando o método proposto. Para ilustrar este fato, é mostrada na Figura 8 a interface da classe *ListComprehension*, introduzida na seção 2.

```
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 8: Interface da classe *ListComprehension*

4 Considerações Finais

Recentemente, tem se notado uma crescente preocupação com os aspectos organizacionais relacionados ao desenvolvimento de compiladores. Mesmo quando são usadas técnicas comprovadamente úteis como a especificação formal de linguagens de programação, parecer haver um consenso a respeito da necessidade de se estruturar melhor estes produtos, como pode ser observado em [6, 10, 11], através do suporte a características como modularidade e estensibilidade, que são centrais durante o ciclo de vida de qualquer programa. Por outro lado, técnicas orientadas a objetos para desenvolvimento de compiladores [2, 9] tratam satisfatoriamente destas características, mas normalmente não suportam o rigor e a corretude necessários. É com este intuito que o método proposto procura obter vantagens no uso do enfoque orientado a objetos em conjunto com especificações formais.

Em linhas gerais, o método se mostrou bastante satisfatório nos casos em que foi aplicado, organizando o processo de desenvolvimento e se adequando ao que é realizado na prática quando necessário. A partir da especificação em BNF da gramática de cada linguagem são derivadas hierarquias de classes organizadas por herança e agregação. Esquemas de tradução são usados na derivação de métodos para cada uma destas classes, explorando o polimorfismo presente tanto na linguagem formal quanto nas linguagens de desenho e implementação. Estas derivações seguem regras precisas, e direcionam a criação de todos os componentes do compilador que dependem da linguagem de programação. Para tratar dos componentes independentes da linguagem, foi definido um *framework* reutilizável (não apresentado aqui).

Para validar este método, foi escolhido o desenvolvimento de um compilador para uma linguagem funcional. Neste foram integradas várias técnicas para compilação de programas funcionais, como a análise de dependências, o casamento de padrões e o *fully lazy lambda-lifting*. A implementação deste compilador (e do interpretador para **GCODE**) totaliza atualmente 16000 linhas de código, escritas usando a linguagem C++ e as linguagens de especificação das ferramentas *lex* e *yacc*. Este produto é portátil, podendo ser executado sobre os sistemas operacionais MS-DOS[3] e UNIX[4]. Tal produto gera um código tão eficiente quanto outras implementações baseadas nas mesmas técnicas, estando organizado de forma mais estruturada e modular, preservando toda a clareza conceitual das especificações.

Uma boa medida para avaliar a estensibilidade dos produtos criados utilizando este método consiste na incorporação de novas funcões a cada compilador. No exemplo apresentado, embora não tenham sido implementadas, técnicas como *type checking* e *strictness analysis* nos moldes propostos em [5] podem ser facilmente introduzidas, conforme delineado em [3], anotando as respectivas árvores com informações apropriadas. Fatos como este, somados à modularidade alcançada, levam a crer que o uso de orientação a objetos de maneira rigorosa permite que aqueles objetivos almejados quando soluções convencionais são usadas de forma isolada possam ser efetivamente alcançados, e em particular no desenvolvimento de compiladores.

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

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

Referências

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

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

[3] C. H. C. Duarte. The development of a Miranda compiler using an object-oriented method. Master's thesis, Department of Informatics, Pontificial Catholic University of Rio de Janeiro, Brazil, July 1994. In Portuguese.

[4] J. A. Goguen and M. Moriconi. Formalisation in programming environments. *IEEE Computer*, pages 55–64, November 1987.

[5] S. L. P. Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[6] S. L. P. 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.

[7] S. L. P. Jones and D. Lester. *Implementing Functional Languages: A Tutorial.* Prentice Hall, 1992.

[8] S. L. P. Jones and P. Wadler. A static semantics for HASKELL. Department of Computing Science, University of Glasgow, May 1991.

[9] K. Koskimies. Software engineering aspects in language implementation. In *Proc. 2^{nd} CCHSC Workshop*, volume 371 of *Lecture Notes in Computer Science*, pages 39–51. Springer Verlag, October 1988.

[10] S. L. Meira, M. A. Musicante, and A. Santos. The design and implementation of language A. In *Proc. V Brazilian Symposium on Software Engineering*, pages 237–256, October 1991. In Portuguese.

[11] N. 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.

[12] B. Stroustrup. *The C++ Programming Language.* Prentice-Hall, 2^{nd} edition, 1992.

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