Um Arcabouço de Compilação Orientado por Aspectos

Mário C. C. Lobato, Roberto S. Bigonha, Mariza A. S. Bigonha

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
{mlobato, bigonha, mariza}@dcc.ufmg.br

Abstract. *This paper describes the implementation of a framework for implementing compilers. This framework is divided in two parts. In the first part, the framework automatically builds lexer, parser and classes of abstract syntax tree nodes. In the second part, the framework uses Aspect-Oriented Programming to implement semantics analysis and intermediate code generation.*

Resumo. *Este artigo descreve a implementação de um arcabouço para implementar compiladores. Este arcabouço é dividido em duas partes. Na primeira parte, o arcabouço gera automaticamente o analisador léxico, o sintático e as classes dos nodos da AST (abstract syntax tree). Na segunda, o arcabouço usa a programação orientada por aspectos para a implementação da análise semântica e geração de código intermediário.*

1. Introdução

Compiladores surgiram no inicio da década de 1950. A construção do primeiro compilador para a linguagem *Fortran* levou 18 homens-anos para ficar pronta [2]. Desde então, o desenvolvimento das teorias de compiladores e a construção de muitas ferramentas conhecidas como *Compiler Compilers*, que automatizam etapas do desenvolvimento, simplificaram essa tarefa. Ferramentas como *Lex* [18], *Yacc* [13], *Flex* [22], *Bison* [4], *Jflex* [16] e *Cup* [1] são capazes de gerar automaticamente analisadores léxicos e sintáticos por meio de definições escritas em declarações de alto nível de abstração.

Os arcabouços (frameworks) [5, 12] de compilação são ferramentas que, além de permitir a geração automática dos analisadores léxicos e sintáticos, conseguem obter reúso de projeto para o desenvolvimento da análise semântica e geração de código. Nesta linha tem-se o *SableCC* [6], que utiliza para a implementação da análise semântica o padrão de projeto Visitor [9]. Um outro arcabouço, o sistema *Polyglot* [20], que tem Java como linguagem base, permite que se altere a sintaxe e a semântica da linguagem base, de modo a permitir que os compiladores gerados para as linguagens similares a Java possam obter o máximo de reúso de código e de projeto da implementação do compilador de Java. Contudo, estes dois sistemas apresentam problemas principalmente em suas metodologias de implementação da análise semântica e geração de código, como será mostrado ao longo deste artigo.

Com os objetivos de facilitar a geração de compiladores para novas linguagens e permitir fácil manutenção dos compiladores quando ocorrem mudanças na definição da linguagem, este artigo apresenta *ACOA*, um arcabouço para implementação de compiladores. A decisão para a construção deste arcabouço baseia-se em dois objetivos: (1) obter

os benefícios dos *compiler compilers* possibilitando a automatização de etapas do desenvolvimento de um compilador; (2) obter um maior grau de abstração e modularização nas etapas implementadas pelo usuário, possibilitando maior extensibilidade dos compiladores desenvolvidos.

O *ACOA* gera compiladores escritos em C++ utilizando a programação orientada por aspectos [15, 26] baseada na linguagem AspectC++ [23] para a implementação da análise semântica e geração de código. Suas principais características são: (a) os analisadores léxico e sintático são gerados automaticamente; (b) as classes dos nodos da AST são geradas automaticamente; (c) o analisador sintático gerado cria automaticamente a AST; (d) a análise semântica pode ser particionada em vários passos de compilação; (e) a implementação dos passos de compilação e a geração de código intermediário [1] são feitos em métodos que são inseridos estaticamente nas classes da AST por meio da inserção estática da programação orientada por aspectos; (f) o uso da programação orientada por aspecto permite obter a modularização das implementações e a total separação do código gerado automaticamente do código escrito pelo usuário.

O uso de *ACOA* para a construção de um compilador é feito do seguinte modo: inicialmente, o usuário constrói um arquivo de especificação que possui, em alto nível de abstração, as definições do analisador léxico, sintático, nodos da AST e as declarações dos passos de compilação. Em seguida o usuário usa este arquivo de especificação como entrada para o gerador de *front-end*, denominado *FrEG* (*Front End Generator*), para que as partes automáticas sejam geradas. O *FrEG* é uma ferramenta proposta neste trabalho. Por fim o usuário implementa os passos de compilação em métodos das classes de nodos da AST gerados. Esses métodos são implementados por meio de aspectos e são inseridos nas classes apropriadas no momento da combinação (*weaving*) dos códigos.

Para facilitar o entendimento de *ACOA* e a demonstração de sua aplicabilidade, as demais seções deste artigo usam como exemplo ilustrativo uma pequena linguagem de programação, chamada *L*, cuja gramática na notação BNF estendida [27] é mostrada na Figura 1.

```
Program  ::= Decl Com
Decl     ::= {Identifier ";"}
Com      ::= {Identifier ":=" Exp ";"}
Exp      ::= ConstantInt | Identifier | Exp "+" Exp
```

Figura 1. Gramática de *L*

2. Arquitetura do Arcabouço

Figura 2 apresenta uma visão geral da arquitetura de *ACOA*. Nela, os elementos que aparecem dentro do retângulo pontilhado são gerados automaticamente ou então já foram previamente implementados. O usuário somente precisa implementar os modulos: *Input File* e *Aspects*.

- ***Input File***: arquivo definido pelo usuário, mostrado na Figura 3. Nesse arquivo são definidos a estrutura léxica e a gramática de *L*, os nodos da AST e

[1] A partir desse ponto do texto, a geração de código intermediário é referenciada apenas como geração de código.

sua construção, e os passos de compilação do compilador de *L* e sua ordem de execução.

- ***Aspects***: representam os aspectos criados pelo usuário. São responsáveis pela inserção estática dos métodos que implementam os passos do compilador nas classes concretas da AST. O usuário pode também opcionalmente inserir outros métodos e membros nas classes abstratas e concretas da AST.

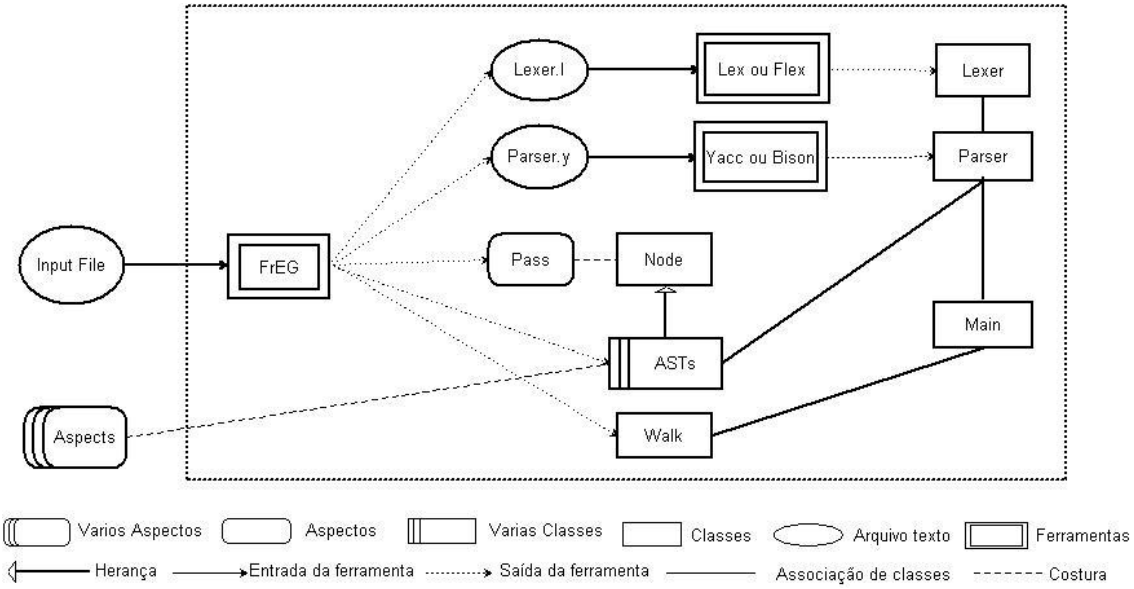

Figura 2. Arquitetura de *ACOA*

Os elementos que compõem a arquitetura de *ACOA* são especificados a seguir, considerando como exemplo a construção de um compilador para a linguagem *L*.

- ***FrEG***: ferramenta que a partir das definições feitas no arquivo de entrada gera automaticamente as seguintes saídas:
  - ***Lexer.l***: definição do analisador léxico de *L* no formato adequado para ser usado pelo *Lex* [18] ou *Flex* [22].
  - ***Parser.y***: definição da gramática de *L* e ações semânticas no formato adequado para ser usado no *Yacc* [13] ou *Bison* [4]. As ações semânticas têm como objetivo a construção da AST.
  - **`Pass`**: Aspecto que introduz estaticamente na classe `Node` os métodos abstratos que correspondem aos passos do compilador de *L*.
  - ***ASTs***: classes abstratas e concretas que representam os nodos da AST de *L*. Todas essas classes são subclasses da classe abstrata `Node`.
  - **`Walk`**: classe que inicia o caminhamento na AST para cada passo do compilador de *L*.
  - **`Factory`**: classe que cria os nodos da AST. Essa classe é usada no analisador sintático (Parser) para construir a AST.
  - ***makefile***: arquivo gerado opcionalmente pelo *FrEG*, e que não é mostrado na Figura 2. O *makefile* contém a combinação e a compilação das classes dos nodos da AST.
- **`Node`**: Classe base de todas as outras classes da AST.

- ***Lex*** ou ***Flex***: Ferramenta que recebe o arquivo *Lexer.l* como entrada e gera como saída o analisador léxico (*Lexer*) de *L* implementado em C++.
- ***Yacc*** ou ***Bison***: ferramenta que recebe o arquivo *Parser.y* como entrada e gera como saída o analisador sintático de *L* (*Parser*) implementado em C++.
- **`Main`**: classe que manipula os arquivos de entrada do compilador de *L* e ativa o analisador sintático e outros passos de compilação, como, análise semântica e a geração de código. A `Main` é opcionalmente gerada pelo *FrEG*, esta flexibilidade permite que o usuário tenha como opção usar e alterar a `Main` gerada ou possa construir a sua própria `Main`.

3. O Gerador de *Front-End* (*FrEG*)

O arquivo de especificação usado como entrada para o *FrEG* é dividido em três partes: (1) definições para o analisador léxico (*lexer*) e definições dos nodos da AST para os *Tokens*; (2) definições para o analisador sintático (*parser*) e dos nodos da AST para as regras gramaticais; (3) declarações dos passos. A Figura 3 mostra o arquivo de especificação da linguagem *L* para o *FrEG*.

```
%%Lexer
letter [A-Za-z_]
number [0-9]
id {letter}({letter}|{number})*
intnum {number}+

%Pattern
{intnum}                        -> ConstanteInt ;
{id}                            -> Identifier;
":="                            -> atrop;
"+"                             -> add;
";"                             -> semicolon;
[\n]                            -> endline;
[ \t\r]                         ;
.                               -> ERROR;

%%Parser
%start Program
%left add

%Grammar
Program ::= Decl Com -> Programa(Decl:decl_programa, Com:com_programa) ;
Decl:Declaracoes ::= Identifier semicolon Decl -> Decl_Var(Identifier:id, Decl:Decl_Cont)
                   | ;
Com:Comandos ::= Identifier atrop Exp semicolon Com
                  -> Atr(Identifier:id, Exp:expr, Com:Com_Cont)
               |  ;
Exp:Expressoes ::=  ConstanteInt -> Numero(ConstanteInt:constante)
                 |   Identifier -> Variavel(Identifier:id)
                 |   Exp1 add Exp2 -> Adicao(Exp1, Exp2);

%%Passes
CadastraVariaveis VerificaVariaveis
```

Figura 3. Especificação de *L* para o *FrEG*

A especificação do analisador léxico começa com a palavra-chave `%%Lexer`, seguida de regras. No exemplo são mostrados dois tipos de regras: sinônimos e padrões. Os sinônimos dão nomes a expressões regulares, por exemplo: `number [0-9]`, de tal forma que a partir dessa declaração pode-se usar o nome `number` entre chaves como uma abreviatura de `[0-9]`.

Os padrões são usados para definir *tokens* e suas ações semânticas. Os *tokens* são definidos via as expressões regulares que ocorrem do lado esquerdo do símbolo `->`, e as ações semânticas aparecem à direita. As ações semânticas, mostradas na Figura 3, são usadas para:

- definir *token* indicador de erro léxico, por exemplo:
  ```
  .          -> ERROR;
  ```
- definir *token* da gramática, por exemplo:
  ```
  "+"        -> add;
  ```
  FrEG usa o nome à direita do símbolo `->` como o nome do terminal que aparecerá nas regras gramaticais;
- adicionar o contador de linhas do código fonte. Por exemplo:
  ```
  [\n]       -> endline;
  ```

Para cada terminal definido, *FrEG* gera a declaração de uma classe de um nodo da AST correspondente ao terminal. Para ilustrar, considere os terminais de *L*, cujas classes são apresentadas na Figura 4.

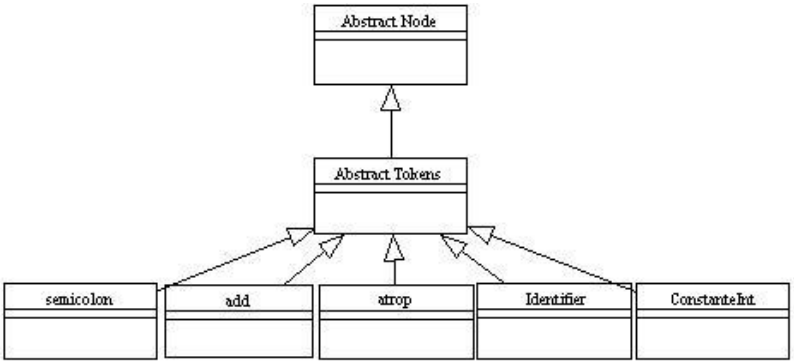

Figura 4. Classes da AST

A especificação do analisador sintático (*parser*) começa com a palavra-chave `%%Parser` seguida por declarações e regras gramaticais. Existem dois tipos de declarações: uma que define a precedência e associatividade dos operadores, e outra que determina o símbolo inicial da gramática. A precedência e associatividade de operadores são definidos pelas palavras-chave `%left`, `%right`, `%nonassoc` seguidas por uma lista de terminais ou nomes. O símbolo inicial da gramática é definido pela palavra-chave `%start`.

Uma regra gramatical para o *FrEG* tem o seguinte formato:

```
Left_Hand ::= Production_Elements -> Ast_Construction ;
```

onde `Left_Hand` representa um não-terminal da gramática, `Production_Elements` representa uma seqüência de zero ou mais terminais ou não-terminais, e `Ast_Construction` representa a ação semântica da regra, que por sua vez determina ao analisador sintático a inserção do nodo declarado na AST.

Um não-terminal é denotado por um `identificador` ou então por um par `identificador : identificador`, onde o segundo identificador especifica o nome alternativo para a classe a ser associada ao não-terminal. *FrEG* gera, a partir dos não-terminais do lado esquerdo de uma produção gramatical, classes abstratas da AST,

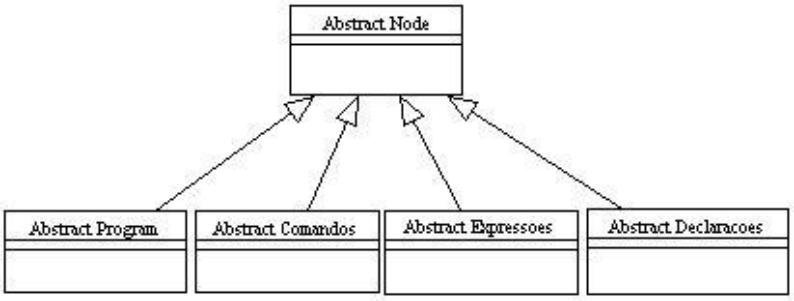

Figura 5. Classes da AST

usando o nome do não-terminal ou o nome alternativo. Essa flexibilidade tem por objetivo permitir que o usuário use um nome mais elucidativo nas classes geradas, enquanto nas regras gramaticais utilizam-se nomes mais simples ou abreviaturas. Para ilustrar, considere os não-terminais da definição de *L*, cujas classes abstratas geradas estão na Figura 5.

FrEG a partir de uma `Ast_Construction`, gera uma classe concreta de um nodo da AST. `Ast_Construction` tem o seguinte formato:

```
Ast_Id ( Ast_Elements )
```

onde `AST_Id` é o nome da classe e `Ast_Elements` é uma seqüência de zero ou mais terminais ou não-terminais, separados por vírgula, que aparecem no `Production_Elements`. Esses elementos são os membros da classe. Os elementos do `Production_Elements` podem ser decorados com índices, os quais não alteram o significado dos elementos, servindo apenas para diferenciar elementos de mesmo tipo no momento de se construir o nodo da AST.

O usuário também pode alterar os nomes dos membros das classes. Para isso basta usar o símbolo "**:**"depois de qualquer elemento contido na `Ast_Elements` e em seguida usar o nome desejado. Desse modo os nomes dos membros ficam mais legíveis e mais simples de serem usados posteriormente. Por exemplo, as classes concretas geradas pelo *FrEG* para a definição de *L* são mostradas na Figura 6. Observe que a classe `Adicao` não tem nenhum dos seus membros com o nome alterado, enquanto a classe `Atr` tem todos os seus membros com nomes alternativos.

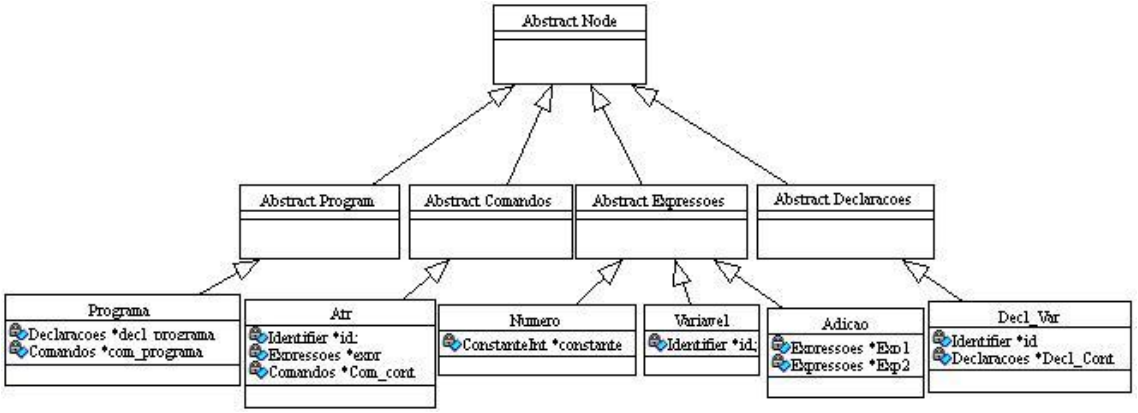

Figura 6. Classes da AST

As classes para os não-terminais são gerados como superclasses das classes concretas, porque alguns dos membros das classes concretas precisam ser polimórficos. Essa necessidade aparece na seguinte regra gramatical de *L*.

```
Com:Comandos ::= Identifier atrop Exp semicolon Com
              -> Atr(Identifier:id, Exp:expr, Com:Com_Cont)
```

O elemento `Exp` que aparece do lado direito de `::=` e antes da regra semântica pode ser qualquer uma das regras:

```
Exp:Expressoes ::= ConstanteInt -> Numero(ConstanteInt:constante)
                |  Identifier -> Variavel(Identifier:id)
                |  Exp1 add Exp2 -> Adicao(Exp1, Exp2);
```

Portanto, o membro `expr` da classe `Atr` mostrado na Figura 6 necessita referenciar qualquer um dos seguintes nodos: `Numero`, `Variavel`, `Adicao`. Isso se torna possível devido ao fato de *expr* ser uma referência para o tipo `Expressoes`, que é a superclasse das classes desses nodos.

A definição e a ordem dos passos de compilação são feitas começando-se pela palavra-chave `%%Passes` seguida por uma lista de nomes dos passos. A ordem em que os nomes aparecem determina a ordem dos passos de compilação. Por meio desta definição *FrEG* gera o aspecto `Pass` e a classe `Walk`.

O aspecto `Pass` introduz estaticamente na classe `Node` métodos abstratos que correspondem aos passos do compilador. Os nomes dos dos métodos inseridos na classe `Node` correspondem aos nomes dos passos de compilação declarados. A classe `Node` é a classe base de todas as outras classes da AST. Logo todas as classes da AST herdam esses métodos inseridos. As classes da AST que são instanciadas (classes concretas) precisam ter esses métodos redefinidos pelo usuário via aspectos. Se o usuário esquecer-se de redefinir qualquer um dos métodos para qualquer uma das classes concretas da AST, o código do compilador gerado pelo arcabouço vai produzir um erro de compilação por existir uma tentativa de instanciar uma classe abstrata.

A classe `Walk` possui um único método chamado `start`, que recebe o nodo raiz da AST e ativa os métodos que implementam os passos para este nodo, iniciando o caminhamento da AST para cada passo do compilador.

4. Implementação dos Passos de Compilação

O objetivo dos aspectos dentro do arcabouço é adicionar novas características às classes da AST. Essas novas características correspondem às implementações dos passos de compilação, por exemplo, a verificação de nomes, a verificação de tipos, a geração de código, etc.

Para ilustrar a implementação dos passos de compilação, a Figura 7 mostra a implementação dos passos `CadastraVariaveis()` e `VerificaVariaveis()` declarados no arquivos de especificação de *L* mostrado na Figura 3. O passo `CadastraVariaveis()` cadastra as variáveis na tabela de símbolos e o passo `VerificaVariaveis()` verifica se as variáveis usadas no programa foram declaradas.

O usuário ao implementar os métodos dos passos de compilação, deve definir primeiro a inserção estaticamente nas classes concretas da AST como mostrado nas li-

```
1  ... //Definições de Bibliotecas e variáveis globais
2  aspect aspectos {
3    public:
4          advice "Programa" : void CadastraVariaveis(){
5              tabela = TabelaSimbolo(); //instância a tabela de símbolos
6              erros = new Erros(); //instância a classe de mensagens de erros
7              if(decl_programa != NULL) //verifica se existe declarações
8                decl_programa->CadastraVariaveis(); //chama o método do nodo filho
9              if(erros->numErros() > 0) {
10               erros->imprime(); //imprime os erros
11               exit(1);
12             }
13         }
14
15         advice "Decl_Var" : void CadastraVariaveis(){
16             if(!tabela->insere(id->getToken_value())) //retorna false se o identificador já foi definido
17               erros->IdentificadorDefinido();
18             if(Decl_Cont != NULL)
19               Decl_Cont->CadastraVariaveis();
20         }
21
22         advice "Atr" || "Numero" || "Variavel" || "Adicao" : void CadastraVariaveis() {}
23
24         advice "Programa" : void VerificaVariaveis(){
25             if(com_programa != NULL) //verifica se existe declarações
26               com_programa->VerificaVariaveis(); //chama o método do nodo filho
27             if(erros->numErros() > 0)
28               erros->imprime();
29             delete tabela;
30             delete erros;
31         }
32
33         advice "Atr" : void VerificaVariaveis(){
34             if(!tabela->pesquisa(id->getToken_value())) //retorna false se o identificador não foi definido
35               erros->IdentificadorNaoDefinido();
36             expr->CadastraVariaveis();
37           }
38
39         advice "Variavel" : VerificaVariaveis(){
40             if(!tabela->pesquisa(id->getToken_value()))
41               erros->IdentificadorNaoDefinido();
42         }
43
44         advice "Adicao" : void VerificaVariaveis(){
45             exp1->VerificaVariaveis();
46             exp2->VerificaVariaveis();
47         }
48
49         advice "Decl_Var" || "Numero" : void VerificaVariaveis(){}
50
51 };
```

Figura 7. Implementação dos passos de compilação de *L*

nhas 4, 15, 22, 24, 33, 39, 44 e 49. Por exemplo, na linha 4 diz ao combinador que o método `CadastraVariaveis` deve ser inserido estaticamente na classe `Programa`. Na implementação dos métodos, é possível usar os membros das classes concretas da AST que foram definidos no arquivo de especificação. Por exemplo, a linha 8 usa o membro `decl_programa`. O usuário também pode fazer uma única implementação para mais de uma classe, como mostrado nas linhas 22 e 49.

Na Figura 8 são esboçadas a inserção estática do aspecto `Pass` na classe `Node` e a inserção estática dos métodos definidos na Figura 7 para a classe concreta `Adicao`.Para entender a Figura 8, considere que os membros e métodos declarados em um aspecto possuem uma ou mais setas mostrando em que classe ou classes eles são inseridos estaticamente. Os membros e métodos que aparecem dentro de um retângulo em uma classe diz em que os mesmos foram inseridos estaticamente por um aspecto.

5. Inserção estática e Padrão Visitor

O objetivo desta seção é mostrar os benefícios do uso da inserção estática da programação orientada por aspectos na implementação dos passos de compilação em vez do padrão de projeto Visitor, muito usado para esse propósito.

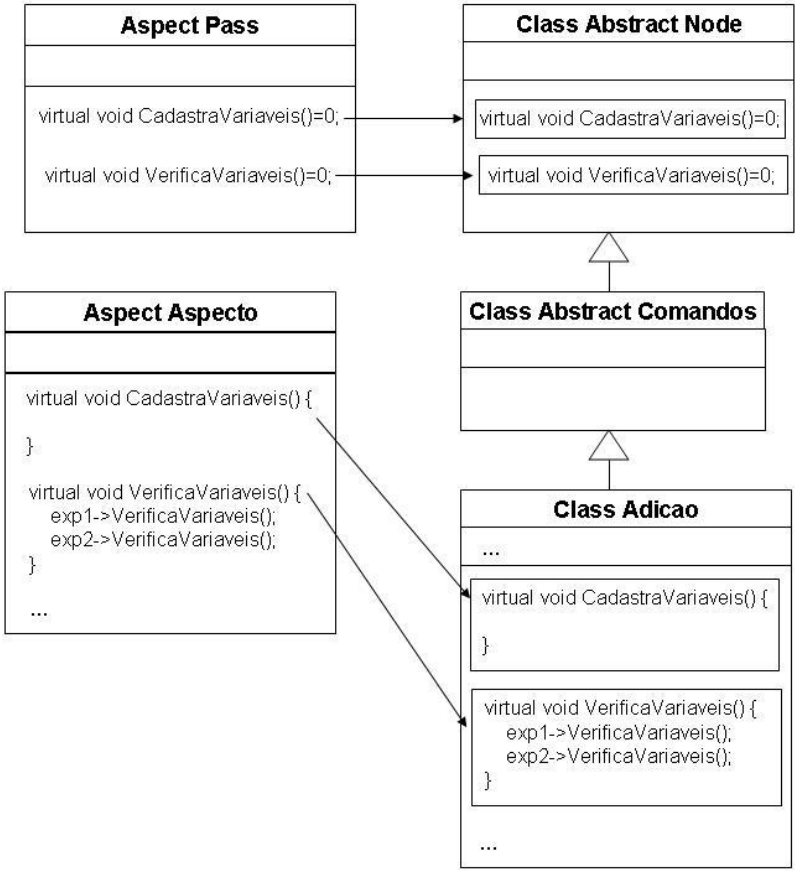

Figura 8. Implementação dos passos de compilação

O padrão Visitor permite adicionar um ou mais comportamentos em uma classe dentro de uma hierarquia. Cada implementação do Visitor descreve um novo comportamento cuja intenção é adicioná-lo em cada classe da hierarquia. Esses novos comportamentos são invocadas por um novo método que é necessariamente adicionado a todas as classes. Este método recebe um objeto do tipo Visitor como parâmetro, e, a partir deste, acessa os novos comportamentos.

A programação orientada por aspectos tem como objetivo permitir a definição separada de interesses transversais (*crosscutting concerns*) às classes de um sistema orientada por objeto. Existem dois tipos de implementação dos interesses transversais: a combinação dinâmica que permite modificar o comportamento da execução do programa, e a combinação estática permite redefinir a estrutura estática das classes, interfaces ou outros aspectos. A inserção estática de constantes, variáveis, atributos, métodos e inicializadores em uma classe no momento da combinação de código é um dos tipos de combinação estática encontrados nas linguagens orientadas por aspectos.

O principal benefício no uso do padrão Visitor está relacionado ao fato de se poder adicionar novos comportamentos a classes de uma hierarquia de classes sem modificá-las, apenas criando novas subclasses do Visitor. Contudo, embora muito úteis, cuidados devem ser tomados no uso do padrão Visitor e de outros padrões de projetos devido aos seguintes problemas [10]:

Problema da Confusão: o uso de padrões torna difícil distinguir entre as definições re-

sultantes da aplicação do padrão daquelas que não resultam deles. Isso pode ser explicado pela necessidade de modificar classes existentes adicionando algo para que sejam aplicáveis os padrões.

Problema da Indireção: o padrão Visitor, assim como outros padrões, usa explicitamente o mecanismo de delegação para ativar comportamentos definidos nas classes do Visitor. Isso torna o programa mais difícil de ser entendido, aumentando a comunicação entre objetos, além de introduzir dependências que podem dificultando algumas evoluções do sistema.

Problema da Quebra de Encapsulamento: o uso do mecanismo de delegação no padrão Visitor também causa alguns problemas relacionados ao encapsulamento. Por exemplo, as classes do Visitor podem precisar usar os atributos das classes nas quais estão adicionando novos comportamentos. Para que o padrão Visitor seja eficiente, esses atributos precisam ser declarados como públicos, ou pelo menos prover algum meio de acesso público para eles. Entretanto pode ser desejável que eles sejam privados ou protegidos.

Problema Relacionado à Herança: aplicar padrões a classes que já possuem uma superclasse pode ser difícil quando a implementação é feita em uma linguagem de programação que não suporta herança múltipla. Além disso, isso aumenta a dependência entre as classes da aplicação e as classes introduzidas durante a aplicação do padrão, dificultando o seu reúso. O padrão Visitor não é diretamente afetado por esse problema, entretanto, se for considerado algumas variações onde os novos comportamentos são adicionados a classes de diferentes hierarquias, esse problema pode ocorrer.

A inserção estática também pode adicionar novos comportamentos a uma hierarquia de classes sem modificá-las como o padrão Visitor. Contudo, o que a torna mais atraente é o fato de se conseguir conseguir resolver os problemas associados ao Visitor, a saber [10]:

Problema da confusão: esse tipo de problema não ocorre com a combinação estática, pois não é necessário adicionar novas operações nas classes para que a inserção estática funcione.

Problema da indireção: é resolvido na inserção estática, pois os métodos são direta e estaticamente introduzidos nas classes as quais eles são aplicados.

Problema da quebra do encapsulamento: não existe na inserção estática porque a delegação não é mais usada.

Problema de herança: apesar de não ser muito danosa para o padrão Visitor, pode ser diminuída usando a inserção estática. Por exemplo, é possível adicionar novos comportamentos às classes de diferentes hierarquias pelo uso de corretas introduções estáticas.

6. Trabalhos Relacionados

O sistema *Polyglot* [20] é um arcabouço cujo objetivo é gerar compiladores para linguagens que estendem ou simplificam a linguagem Java de modo que não exista duplicação de código e que possua a extensibilidade escalável. Extensibilidade escalável é definida como uma extensão que deve exigir esforços de programação proporcional à diferença entre a linguagem estendida e a linguagem base. Adicionar novos nodos da AST ou novos passos de compilação demanda a escrita de código do tamanho proporcional às mudanças.

Embora a extensibilidade escalável seja desejável, a mesma e a não duplicação de código fazem com que a estrutura de código do *Polyglot* e o seu mecanismo de extensibilidade sejam complexos e difíceis de entender. O mecanismo de delegação que permite tanto a extensibilidade escalável como a não duplicação de código apresenta o *Problema da Indireção* e o *Problema da Quebra de Encapsulamento* mostrados anteriormente.

O sistema *JastAdd* [11] é um arcabouço para especificação e implementação de compiladores e que gera automaticamente as classes da AST, permitindo que as análises semânticas e a geração de código sejam implementadas convenientemente via AST. As análises semânticas e a geração de código estão em diferentes módulos que são combinados junto às classes da AST usando a técnica de programação orientada por aspectos. *JastAdd* permite que o usuário escolha a ferramenta de geração de analisadores léxicos e sintáticos. Porém é de inteira responsabilidade do usuário a integração das classes de nodos da AST, gerados pelo *JastAdd*, com a ferramenta escolhida para que a AST seja construída. O sistema não utiliza nenhuma linguagem orientada por aspectos para Java, como *AspectJ* [14, 17], ignorando as vantagens que tal uso poderia proporcionar.

O sistema *SableCC* [6], como o *JastAdd*, é um arcabouço para a geração de compiladores implementados em Java para linguagens em geral. *SableCC* possui um arquivo de entrada com as definições léxicas e as produções gramaticais, a partir das quais são gerados os analisadores léxico e sintático respectivamente. Essas definições, diferente das definições de outras ferramentas, não possuem uma ação semântica associada a um token ou a uma produção. Isso faz com que o *SableCC* determine as ações semânticas para os analisadores léxico e sintático para a criação da AST usando apenas as definições contidas no arquivo. As classes da AST são também geradas automaticamente pelo arcabouço usando apenas as definições dos *tokens* e das produções gramaticais. Este fato tem a vantagem de evitar erros provocados pelo usuário se o mesmo tivesse a possibilidade de especificar as classes da AST e o modo de sua construção. Por outro lado, esta facilidade proporciona uma flexibilidade menor para o usuário, que fica obrigado a usar as classes definidas pelo SableCC.

O *SableCC* gera também, a partir das definições do arquivo de entrada, classes relacionadas ao padrão Visitor e ao caminhamento da AST. O usuário é responsável pela implementação das classes de análise semântica e da geração de código usando herança das classes geradas. O sistema usa uma versão estendida do padrão Visitor proposto em [9], mas sua extensão não resolve os problemas apontados na seção anterior.

7. Avaliação e Validação

O *ACOA* é um arcabouço de fácil utilização além de possuir facilidades ausentes nos demais arcabouços descritos. Sua utilização foi mostrada usando como exemplo a linguagem *L*. Dentre suas facilidades oferecida pelo arcabo uço destacam-se:

- o arquivo de entrada para o *FrEG* possui um nível maior de abstração que os correspondentes aos geradores de compiladores existentes;
- o usuário não precisa ler as classes geradas para conhecer os dados que ela possui;
- as partes geradas automaticamente são totalmente integradas ao sistema não havendo necessidade do usuário alterá-las;
- o código criado pelo usuário para a inserção estática nas classes da AST é integrado ao sistema pelo combinador de AspectC++ de modo totalmente automático;

- o usuário pode organizar os aspectos de forma flexível, por exemplo, um aspecto pode armazenar apenas os métodos inseridos em uma classe da AST, ou um aspecto pode armazenar os métodos de um mesmo passo de compilação para todas as classes da AST, etc.

A modularização do sistema é uma das principais vantagens do *ACOA*. A modularização aumenta a extensibilidade do mesmo e permite que alterações na definição da linguagem tenham o menor impacto possível no código criado pelo usuário. Exemplos:

- a criação de um novo passo de compilação, a alteração na ordem de execução dos passos ou a remoção de algum passo podem ser realizados sem a necessidade de qualquer alteração do código de passos já implementados. Ao criar um novo passo, o usuário precisa apenas implementar o novo método para cada nodo da AST;
- a introdução de novas produções na gramática da linguagem que não afetam semanticamente outras produções da mesma pode ser feito sem nenhuma alteração do código anteriormente implementado. Basta que o usuário implemente os passos de compilação para os novos nodos da AST. Nos casos onde as novas produções da gramática alterem semanticamente outras já existentes, é possível saber exatamente os locais que devem sofrer essas alterações, pois as implementações estão modularmente separadas;
- a remoção de produções da gramática tem as mesmas características da criação de novas produções;
- alteração de algum passo de compilação para algum nodo da AST demanda alteração apenas no método que possui a implementação deste passo para este nodo. Não é preciso alterar as implementações para os demais passos deste nodo ou a implementação para os outros nodos.

Com o objetivo de validação, o *ACOA*, foi usado na a implementação de um compilador para a linguagem Machĭna [3, 24, 25]. Machĭna é uma linguagem de especificação ASM [7, 8], que foi desenvolvida no Departamento de Ciência da Computação da UFMG. Com o arcabouço foram desenvolvidas a análise léxica, sintática, semântica e geração de código intermediário. O código intermediário gerado foi MIR (*Machĭna Intermediate Representation*) [19, 21]. A partir deste ponto, o código MIR gerado é usado em outro arcabouço, chamado de kℓar [19], que faz otimizações neste código e gera código C++.

Machĭna é uma linguagem extensa, possuindo 383 produções em sua gramática. O analisador sintático de Machĭna foi gerado sem nenhum conflito *shift-reduce* ou *reduce-reduce*, usando a precedência de operadores, considerando que na gramática de Machĭna as operações binárias e unárias são ambíguas.

Sua especificação para o *FrEG* gerou 644 classes de nodos da AST contando todos os tipos de classes. Foram implementados 4 passos de compilação para o compilador de Machĭna.

Aproximadamente 25.000 linhas códigos foram implementadas manualmente e 36.000 linhas de código foram geradas automaticamente. Portanto, cerca de 59% das linhas de código do compilador de Machĭna foram geradas automaticamente pelo *ACOA*. Isso mostra um ganho de tempo se o mesmo compilador fosse implementado utilizando apenas uma ferramenta *compiler compiler*.

8. Conclusão

Esse artigo apresentou as principais características do *ACOA* que utiliza a programação orientada por aspectos para implementação de análise semântica e geração de código. O arcabouço proposto é uma ferramenta de projeto de linguagens bastante simples de ser usada. Um dos fatores a que se deve a sua simplicidade é a geração automática do analisador léxico, analisador sintático e das classes dos nodos da AST por meio especificação de alto nível para o *FrEG*, o que permite ao desenvolvedor fazer constantes modificações na sintaxe de sua linguagem.

A utilização da inserção estática de métodos e membros em classes da programação orientada por aspectos é outro fator positivo do *ACOA*. A inserção estática apresenta os mesmos benefícios do padrão de projeto Visitor e, mas, não causa os problemas encontrados na solução que usa o Visitor.

Referências

[1] C. S. Ananian, A. Appel, F. Flannery, S. E. Hudson and D. Wang, CUP LALR parser generator for Java, 1996. http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[2] J.W. Backus, R. J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I Ziller, R. A. Hughes, and R. Nutt(1957). *The FORTRAN automatic coding system*. Western Joint Computer Conference.

[3] R. S. Bigonha, F. Tirelo, V. O. Di Ioro and M.A.S. Bigonha, *A Linguagem de Especificação Formal Machina 2.0*, RT 001/2005, LLP/DCC/UFMG, 2005.

[4] C.Donnelly, R.Stallman, *Bison, The Yacc-Compatible Parser Generator*, Version 1.25, ftp://prep.ai.mit.edu/pub/gnu.

[5] M. E. Fayad and D. C. Schmidt, *Object-Oriented Application Frameworks*. Communication of the ACM, 1997

[6] E. M. Gagnon and L. J. Hendren, *SableCC, an Object-Oriented Compiler Framework*. Technical Report School of Computer Science, McGill University, Quebec, Canada. 1998.

[7] Y. Gurevich, *Evolving Algebras. A Tutorial Introduction*. Bulletin of EATCS, 43:264-284, 1991.

[8] Y. Gurevich, *Evolving Algebras 1993: Lipari Guide*. In E. Börger, editor, Specication and Validation Methods, pages 9-36. Oxford University Press, 1995.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Padrões de Projeto, Bookman, 2000.

[10] O. Hachani, D. Bardou, *Using Aspect-Oriented Programming for Design Patterns Implementation*. Position paper at the Reuse in Object-Oriented Information Systems Design workshop. 8th International Conference on Object-Oriented Information Systems (OOIS 2002), Montpellier, France Sept. 2-5 2002.

[11] G. Hedin, E. Magnusson, *The JastAdd system - an aspect-oriented compiler construction system*, SCP - Science of Computer Programming, 47(1):37-58. Elsevier. November 2002.

[12] R. E. Johnson, *Components, Frameworks, Patterns*. Communication of the ACM, 1997.

[13] S. C. Johnson, *YACC - yet another compiler compiler*. Technical Report Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. C. Griswold, *An Overview of AspectJ*. 15th European Conference on Object-Oriented Programming, pages 220-242, LNCS, Vol.1241, Springer-Verlag, June 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier and J. Irwin, *Aspect-Oriented Programming*. proc. 11th European Conference on Object-Oriented Programming, pages 220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[16] G. Klein, *JFlex the fast scanner generator for Java*, 2001. http://www.jflex.de/.

[17] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Company. 2003.

[18] M. E. Lesk, *Lex - A Lexical Analyzer Generator*. Technical Report Computing Science Technical Report 39, AT&T Bell Laboratories, 1975.

[19] K. Magnani, M. A. S. Bigonha, R. S. Bigonha, F. F. Oliveira and V. O. D. Iorio, *An Infrastructure for Implementing Compilers for Concurrent Abstract State Machine Languages*. CLEI'2005, Cali, October, 2005.

[20] N. Nystrom, M. R. Clarkson and A. C. Myers, *Polyglot: An Extensible Compiler Framework for Java*. Proceedings of the 12th International Conference on Compiler Construction, Warsaw, Poland, April 2003. LNCS 2622, pages 138 - 152.

[21] F. F. Oliveira, K. Magnani, M. A. S. Bigonha and R. S. Bigonha, *MIR: Machĭna Intermediate Representation*. Technical Report RT001/04, Laboratório de Linguagem de Programação - DCC, UFMG, 2004.

[22] V. Paxson. *Flex, A fast scanner generator*, Edition 2.5, for flex version 2.5, Free Software Foundation, Inc., März 1995.

[23] O. Spinczyk, A. Gal and W. Schröder-Preikschat, *AspectC++: An Aspect-Oriented Extension to the C++ Programming Language*. In Proceedings of the Fortieth International Conference on Tools Pacific, pages 53-60. Australian Computer Society, Inc. , 2002.

[24] F. Tirelo, M. A. Maia, R. S. Bigonha, and V. O. Di Iorio, *Machĭna: A Linguagem de Especificação de ASM*, Technical Report LP08/99, 40 pages, Laboratório de Linguagens de Programação - DCC UFMG, agosto de 1999.

[25] F. Tirelo, *Uma Ferramenta para Execução de um Sistema Dinâmico Discreto Baseado em Álgebras Evolutivas*. Dissertação de Mestrado, DCC, UFMG, 2000.

[26] F. Tirelo, R. da S. Bigonha, M. da S. Bigonha, and M. T. de O. Valente, *Desenvolvimento de Software Orientado por Aspectos*. XXIII Jornada de Atualização em Informática, XXIV Congresso da Sociedade Brasileira de Computação, August 2004.

[27] N. Wirth, *What Can We Do About the Unnecessary Diversity of Notation for Suntatic Definition?* Communication of the ACM, 20(11).822-823, 1977.