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

Um Arcabouço Para Compilação de Linguagens de Especificação ASM

Mário Celso Candian Lobato

Orientador: Roberto da Silva Bigonha

Co-Orientadora: Mariza Andrade da Silva Bigonha

Belo Horizonte
23 de Março de 2006

Mário Celso Candian Lobato

Um Arcabouço Para Compilação de Linguagens de Especificação ASM

Dissertação apresentada ao Curso de Pos-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
23 de Março de 2006

Universidade Federal de Minas Gerais

Folha de Aprovação

Um Arcabouço Para Compilação de Linguagens de Especificação ASM

Mário Celso Candian Lobato

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Roberto da Silva Bigonha
Departamento de Ciência da Computação - UFMG

Profa. Mariza Andrade da Silva Bigonha
Departamento de Ciência da Computação - UFMG

Prof. Vladimir Oliveira Di Iorio
Departamento de Informática - UFV

Prof. Marcelo de Almeida Maia
Departamento de Ciência da Computação - UFU

Belo Horizonte 23 de Março de 2006

Resumo

O arcabouço *ACOA* é um arcabouço para implementar compiladores. O *ACOA* gera automaticamente o analisador léxico, o sintático e as classes dos nodos da árvore de sintaxe abstrata (AST) e usa a programação orientada por aspectos para a implementação da análise semântica e geração de código. O objetivo do *ACOA* é ser de fácil utilização para a construção de compiladores e permitir fáceis alterações nos compiladores implementados pelo arcabouço.

Abstract

The ACOA framework is a framework for implementing compilers. The ACOA automatically builds lexer, parser and classes of abstract syntax tree nodes (AST) and uses Aspect-Oriented Programming to implement semantics analysis and code generation. The objective of the ACOA is to be of easy use for implementing compilers and to allow easy alterations in the compilers implemented by means of the framework.

Agradecimentos

Primeiro, gostaria de agradecer ao professor Bigonha a orientação, a paciência, os ensinamentos que sem dúvida foram fundamentais para a realização deste trabalho. Gostaria também de agradecer a professora Mariza, principalmente as suas revisões.

Agradeço ao meu pai e minha mãe por todo o apoio, todo o carinho, todo o amor e por todo o esforço que fizeram para que eu pudesse estar aqui. Agradeço às minhas irmãs a companhia e ter-me aturado esses anos aqui em Belo Horizonte.

Agradeço ao meu grande amigo André, companheiro de batalhas desde da época de Viçosa, a amizade e o companheirismo. Agradeço aos meus companheiros do laboratório de linguagens a companhia, a ajuda e principalmente a amizade.

Por fim, gostaria de agradecer à CAPES a ajuda financeira.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Esta dissertação propõe um arcabouço para implementar compiladores. Este arcabouço tem como objetivo de facilitar a geração de compiladores para novas linguagens, e permitir fácil manutenção dos compiladores quando ocorrem mudanças na definição da linguagem.

Este capítulo introduz o conceito de ASM e a linguagem Machĭna, onde o arcabouço é aplicado. Este capítulo levanta os problemas existentes nos ambientes de execução ASM e propõe soluções para resolver esses problemas.

1.1 Máquinas de Estado Abstratas

O modelo de Máquinas de Estado Abstratas (ASM, *Abstract State Machines*) é um modelo formal de especificação e modelagem de sistemas, introduzido por Yuri Gurevich [32, 33] na década de 1990.

A metodologia ASM provê recursos expressivos para especificar a semântica operacional de sistemas dinâmicos discretos, em um nível de abstração natural e de uma maneira direta e essencialmente livre de codificação [18], diferentemente da Máquina de Turing, que necessita uma longa seqüência de passos da máquina para simular um único passo do algoritmo. Um dos objetivos de ASM é diminuir a distância que há entre os modelos formais de computação e os métodos práticos de especificação [33].

Uma especificação ASM contém a definição de um estado inicial, S_0, e uma regra de transição, R, que define as mudanças de estado. A execução de uma especificação é uma seqüência de estados, onde um estado S_i é obtido executando a regra R em S_{i-1}.

Um vocabulário Υ é um conjunto de nomes de funções e relações, cada nome com uma aridade fixa associada. Os nomes de relações de zero argumentos *true*, *false*, o nome de função de zero argumento *undef*, os operadores

booleanos usuais e o sinal de igualdade estão presentes em todo vocabulário.

Um estado S de vocabulário Υ é um conjunto X, denominado o superuniverso de S, junto com as interpretações, em X, dos nomes de funções e relações pertencentes a Υ. Se f é um nome de função de aridade r, então f é interpretado como uma função $\mathbf{f} : X^r \rightarrow X$. Se f é um nome de relação de aridade r, então f é interpretado como uma função $\mathbf{f} : X^r \rightarrow \{true, false\}$. Se U é um nome de relação pertencente a Υ, então o conjunto $U(\bar{x}) = true$ e $\bar{x} \in U$ são proposições equivalentes.

Uma função, ou relação, pode ser estática ou dinâmica. Uma função é dinâmica quando ao mudar de estado, a sua interpretação pode ser modificada em algum ponto. Caso contrário, a função é estática. Basicamente, o conceito de dinâmico do sistema é modelado pelas alterações, de estado para estado, na interpretação de funções dinâmicas.

Uma regra de transição de ASM tem a aparência de um programa escrito em uma linguagem imperativa. A diferença principal é a ausência de comando de iteração, pois este conceito está implícito no mecanismo de execução da máquina. A execução da máquina consiste em processar as regras de transição repetidamente, modificando de cada vez o estado atual.

As regras de transição mais simples de ASM são *atualização*, *condicional* e *bloco*. A regra de atualização tem a forma: $f(\bar{x}) := y$, onde o comprimento de $\bar{x}$ é igual à aridade da função *f*. Esta regra cria, a partir de um estado S, um novo estado *S'*, tal que a interpretação de nome f é uma função, que no ponto $\bar{x}$, o seu valor é *y*. Por exemplo, a regra $f(1) := 2$ determina um novo estado no qual o valor da função *f*, no ponto 1, é 2. Uma regra condicional tem a forma: **if** g **then** R_1 **else** R_2 onde g é uma expressão booleana. Se g for avaliado como verdadeiro, então o estado resultante é o resultado da regra R_1, caso contrário o estado resultante é o resultado da regra R_2. A regra de bloco tem a forma: $R_1, ..., R_n$ onde o estado resultante é formado pelo resultado da execução de todas as regras R_i, em paralelo. Por exemplo, a execução da regra de bloco:

$$f(1) := 2, f(2) := 4$$

produz um novo estado, no qual o valor da função f no ponto 1 é 2 e no ponto 2 é 4.

Além das regras básicas, há também as regras que utilizam variáveis. Variáveis são símbolos que podem denotar elementos do superuniverso. Em ASM, variáveis são utilizadas para modelar paralelismo, não-determinismo e a "criação" de novos elementos. As regras que utilizam variáveis são as regras *import*, *choose* e *forall*. A regra *import* tem a forma:

$$\textbf{import } v \textbf{ do } U(v) := trueR_0 \textbf{ end}$$

onde v é uma variável e R_0 é uma regra. O efeito dessa regra é executar R_0 em um estado em que a variável v está associada a um valor importado de um universo especial chamado *Reserve*. Este universo está contido em X – o superuniverso dos estados de máquina – e contém os elementos que serão importados. Em geral, a regra *import* é utilizada para estender o universo, isto é, adicionar elementos aos universos. A regra *choose* tem a forma:

choose v **in** U **satisfying** g **do** R_0 **end**

onde v é uma variável, U é o nome de um universo finito, g é uma expressão booleana e R_0 é uma regra de transição. O efeito desta regra é executar R_0 em um estado no qual a variável v está associada a um valor pertencente ao universo U. Este valor é escolhido de maneira não-determinista e satisfaz a guarda g. A regra *forall* tem a forma:

forall v **in** U **do** R_0 **end**

onde v é uma variável, U é o nome de um universo finito e R_0 é uma regra de transição. O efeito desta regra é criar uma instância de R_0 para cada elemento pertencente ao universo U. Em cada instância de R_0, a variável v está associada ao elemento correspondente de U. Depois de criadas as instâncias, todas são executadas em paralelo.

ASM possui ainda recursos para expressar paralelismo assíncrono, conhecido como *ASM Multiagente* ou *ASM Distribuída*. Uma *ASM Multiagente* contém um número finito de agentes computacionais que executam concorrentemente um número finito de programas.

Um aspecto importante que deve ser modelado na especificação de um sistema é que, em geral, sistemas são afetados pelo ambiente. O modelo ASM supõe que o ambiente se manifeste por meio de funções denominadas *funções externas*. Um exemplo de função externa é uma entrada fornecida pelo usuário. Pode-se pensar em funções externas como *oráculos*, tais que, a especificação fornece argumentos, e o *oráculo* fornece resultado [33].

Uma das grandes vantagens do modelo é a possibilidade de se executar uma especificação, o que pode facilitar a tarefa de encontrar erros. O modelo possui também recursos para modelar concorrência e não-determinismo. Além disso, pode-se citar também a existência de uma teoria matemática subjacente, a teoria de Álgebra Evolutiva, que permite a prova de propriedades da especificação.

Para maiores detalhes de uma especificação ASM e sua execução, ou sobre *ASM Multi-Agentes*, consultar uma das seguintes referências [18, 32, 33, 67].

1.2 A Linguagem Machĭna

A linguagem Machĭna foi desenvolvida no Departamento de Ciência da Computação da UFMG no ano de 1999 [68, 69]. A primeira versão de Machĭna sofreu alterações e inclusões de novas características dando origem em 2005 à versão 2.0 da linguagem [8].

A linguagem Machĭna é baseada no conceito de Máquinas de Estado Abstratas (ASM), é fortemente tipada e possui suporte à modularidade, criação de novos tipos e construções de alto nível. Um programa em Machĭna consiste na definição de um vocabulário, do estado inicial e da regra de transição que promove a mudança de estado.

Uma das principais características de Machĭna é a modularidade. Um módulo de programa especifica a regra de transição que um agente a ela associado executa, seu vocabulário, isto é, o conjunto de símbolos que manipula, a interpretação destes símbolos no estado inicial e o invariante de execução.

O invariante de execução é uma condição que deve ser satisfeita no início e no fim de todo passo de execução da regra de transição de um módulo. Execuções que não satisfazem o invariante em algum momento são consideradas execuções inválidas do módulo. Isto auxilia a prova de propriedades do sistema especificado, tomando-se somente execuções válidas da máquina.

Outra característica de um módulo de Machĭna é o mecanismo de controle de visibilidade, que permite organizar o vocabulário de um agente em unidades encapsuladas. Toda declaração é automaticamente privada ao módulo no qual foi declarada. Um elemento só não é privado se for explicitamente declarado como público. Os elementos públicos de um módulo *M1* pode ser visível a outro módulo *M2*, se *M2* incluir o vocabulário de *M1*.

Machĭna é fortemente tipada, com um rico conjunto de tipos pré-definidos compostos pelos tipos básicos, tipos compostos e tipos genéricos.

Os tipos básicos são os tipos dos booleanos, caracteres, inteiros, fracionários de ponto flutuante e o das cadeias de caracteres. Os tipos compostos são união disjuntas, arquivos, listas, conjuntos, agentes, tuplas, nodos, enumeração e funcionais. Os genéricos são listas genéricas (lista com elementos de qualquer tipo), agentes genéricos (agentes de qualquer tipo), conjuntos genéricos, e o tipo *?* que é a união disjunta de todos os tipos.

Machĭna permite que o usuário defina os seus tipos fazendo uma composição de tipos pré-definidos ou mutuamente recursivos.

Outro recurso de Machĭna são as abstrações de regras. Abstrações de regras é um recurso apropriado para se definir operações de tipos abstratos de dados. Machĭna possui dois tipos de abstrações de regras: *single-iteration* e *multi-iteration*. Ambas são abstrações parametrizadas de uma regra de

transição. A única diferença é que, quando ativado uma *multi-iteration*, seu corpo é executado repetidamente, como uma sub-máquina, até que uma regra de retorno seja encontrada. No caso de *single-iteration*, executa-se uma única vez a regra de transição do seu corpo. Durante a execução de uma ação, as alterações de estado têm efeito apenas local. Somente após o retorno, o efeito da ação se faz sentir externamente.

As regras de transição de ASM mostradas na Seção 1.1 são todas implementadas na linguagem Machĭna, que adiciona outras regras para facilitar a tarefa de especificação. As regras de transição de Machĭna podem ser dos seguintes tipos: (i) Regras Básicas: Atualização, Bloco e Abreviaturas; (ii): Regras Condicionais: If, Case e With; (iii) Regras de Universalização; (iv) Chamadas de Abstração; (v) Regras de Manipulação de Agentes.

Machĭna também possui recursos para a implementação de *ASM Multi-agentes*. Uma especificação *Machĭna Multiagentes* pode conter um número finito de agentes computacionais que executam concorrentemente um número finito de programas. As comunicações entre agentes são feitas via chamadas de abstrações de regras que são anunciadas em uma interface dos módulos principais dos agentes. Esse procedimento é entendido como envio de mensagem. Uma especificação mais detalhada sobre Machĭna foge do escopo desse texto, podendo ser encontrada em [8].

1.3 Definição do Problema

A especificação ASM foi introduzida em [32, 33] e deste então a literatura apresenta vários exemplos de sua utilização na especificação formal de sistemas, dentre os quais destacam-se:

- Arquitetura de computadores [9, 10, 13].
- Linguagens de programação [14, 15, 16, 34, 72].
- Sistemas distribuídos [6, 7, 11, 35].
- Tempo real [31, 36].

Com o sucesso de ASM, vários ambientes de execução de ASM surgiram para auxiliar as especificações e permitir sua execução. Por exemplo, *Michigan*[41], *ASM-Workbench* [19, 20], *Xasm* [3], *EvADE* [71], *Montages* [2, 47], *AsmGofer* [62], *ASML* [53, 54], *ASM/VA* [26], entre outros.

Del Castillo [21] divide as ferramentas ASM em duas classes:

- Ferramentas que dão suporte ao usuário durante o processo de desenvolvimento da especificação ASM, por exemplo, editores, analisadores estáticos, interpretadores, depuradores. Essas ferramentas auxiliam a especificação ASM como o resultado de um processo iterativo.

- Ferramentas que transformam especificações ASM em outras linguagens de forma a permitir o processamento dessa especificação por outras ferramentas. Esse tipo de ferramenta capacita o uso de compiladores que produzem código eficiente ou ferramentas que permitem a verificação de propriedades da especificação.

Apesar dos esforços de muitos pesquisadores e a melhoria obtida nos últimos anos, o atual estado da arte de ferramentas de suporte para especificação ASM ainda não é satisfatório [21].

Por um lado, existem vários simuladores ASM, a maioria baseada em interpretação, que apesar de executar a especificação ASM não possuem nenhuma outra característica adicional. As possibilidades de interação são em geral poucas, o que não os tornam muito conveniente como ferramentas de desenvolvimento. Por outro lado, quase não existem ferramentas de transformação, isto principalmente devido à necessidade de grandes esforços na sua implementação. Isto posto, acredita-se que é possível diminuir esses esforços por meio do uso de um arcabouço no processo de desenvolvimento dessas ferramentas.

Outra questão importante é o fato de alguns pontos de ASM ainda não estarem totalmente consolidados, como o mecanismo de concorrência. Portanto, seria adequado ter um ambiente que permitisse alterar algumas definições da linguagem de especificação ASM sem que haja a necessidade de se implementar novamente todo o ambiente.

Para suprir as deficiências relacionadas nesta seção, foi desenvolvido um sistema de especificação ASM do qual faz parte a ferramenta desenvolvida nesta dissertação e apresentada neste texto. Esse sistema tem como objetivos:

- Diminuir os esforços de implementação de um compilador que transforme as especificações ASM em código C++.

- Permitir que o desenvolvedor altere características de sua linguagem de especificação ASM com a garantia de que o impacto dessas alterações seja o menor possível no código anteriormente implementado.

- Ter à disposição um conjunto de otimizações feitas para ASM.

- Poder definir e experimentar novas otimizações do código gerado para especificações ASM.

- Utilizar otimizações de código promovidas pelo compilador C++.

1.3.1 Sistema de especificação ASM

O sistema de especificação ASM é conhecido como *Projeto Machĭna* e sua especificação original está dividida em três partes, como mostra a Figura 1.1. A primeira parte compreende o *front-end* de Machĭna, que transforma a definição de Machĭna, para o código intermediário MIR (*Machĭna Intermediate Representation*) [57, 50]. A segunda parte recebe o arquivo MIR, fazendo otimizações nesse código. A terceira parte transforma MIR em código C++.

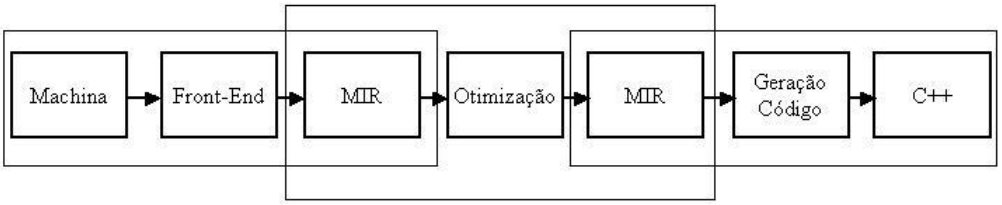

Figura 1.1: Projeto Machĭna (versão original)

O *projeto Machĭna* cresceu, evoluindo para um ambiente de desenvolvimento de projeto para qualquer linguagem de especificação ASM, permitindo que sejam realizados testes e experiências sobre a linguagem. A Figura 1.2 esquematiza o atual *Projeto Machĭna* mostrando os três projetos que o compõe:

- Arcabouço de compilação do *front-end*, objeto dessa dissertação.
- Arcabouço de geração de código, *back-end*, objeto de uma outra dissertação [50].
- Conjuntos de otimizações do *front-end*, objeto de uma tese de doutorado [57].

O arcabouço de compilação do *front-end* proposto nesta dissertação tem como objetivo facilitar:

- a geração de compiladores para linguagens de especificação ASM;
- alterações em compiladores anteriormente desenvolvidos pelo arcabouço quando a linguagem de especificação ASM tem sua definição alterada (sintaxe ou semântica) de modo a aproveitar o máximo de código anteriormente implementado.

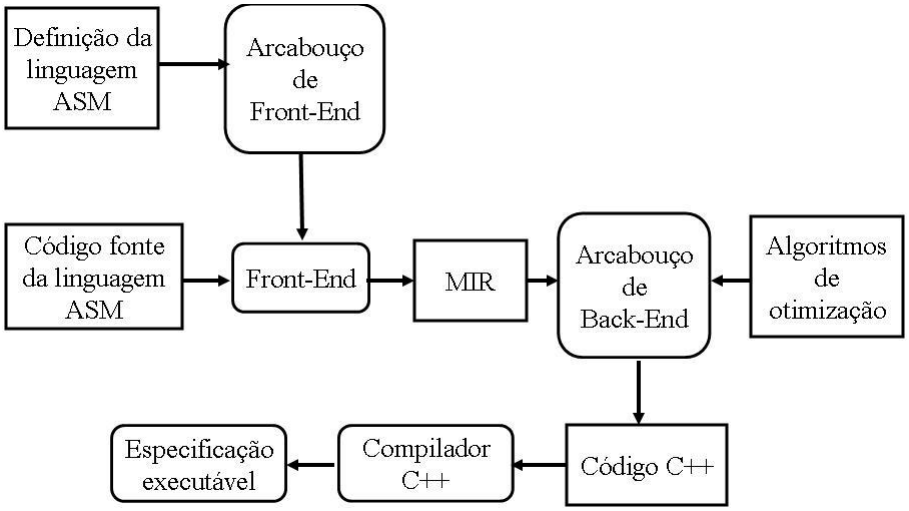

Figura 1.2: Projeto Machĭna (nova versão)

Essas facilidades podem ser alcançadas devido ao fato do arcabouço possuir as seguintes características: (1) a automatização de etapas do desenvolvimento de um compilador por meio de uma especificação em alto nível de abstração usada como entrada para o gerador de *front-end*; (2) a especificação para o gerador de *front-end* é definida em uma linguagem mais abstrata do que a maioria das linguagens usadas nas ferramentas *Compiler-compilers* [43, 49, 59, 25, 46, 1], permitindo que o usuário concentre mais nos detalhes léxico e sintático de sua linguagem do que na implementação do compilador; (3) as etapas implementadas pelo usuário obtém maior extensibilidade, flexibilidade e modularização por meio do uso da programação orientada por aspectos [45, 70], possibilitando maior evolução dos compiladores desenvolvidos, como é mostrado no Capítulo 4.

Os compiladores gerados pelo arcabouço aceitam como entrada uma especificação escrita na linguagem definida pelo usuário e têm como saída este mesmo programa no código intermediário MIR. Esse código é a entrada do arcabouço do *back-end*.

MIR originalmente foi usada para a representação intermediária da linguagem Machĭna, mas hoje ela consegue representar qualquer linguagem concorrente ASM. MIR foi projetada com o objetivo de facilitar otimizações de código, o que permite o desenvolvimento de otimizações especificas para o modelo ASM.

O arcabouço de geração de código é chamado kℓar [50]. Além de gerar código C++ a partir do código MIR, kℓar disponibiliza um ambiente que permite a inclusão de algoritmos de otimizações. Essas otimizações são es-

pecificas para o modelo de ASM e fazem parte do terceiro projeto [56]. Esta facilidade permite que o usuário do sistema construa e use suas próprias otimizações.

Por fim, o código C++ gerado pelo kℓar é compilado por um compilador C++. Ressalta-se que as otimizações do terceiro projeto não superpõem as otimizações convencionais, sendo estas deixadas a cargo do compilador que compila o código C++ gerado.

1.3.2 Validação do Arcabouço Proposto

O arcabouço de front-end proposto por essa dissertação é validado via a implementação do compilador para a linguagem Machĭna, que foi apresentada na Seção 1.2.

1.4 Estrutura da Dissertação

Esta dissertação está organizada em seis capítulos e dois apêndices. Este capítulo introduziu o conceito de ASM, a linguagem Machĭna, levanta os problemas existentes nos ambientes de execução ASM e propõe soluções para resolver esses problemas.

Capítulo 2 faz uma revisão bibliográfica, apresentando os principais conceitos relacionados ao trabalho proposto.

Capítulo 3 apresenta três ferramentas de compilação: *Polyglot*, *JastAdd* e *SableCC*. Para cada uma são mostrados suas arquiteturas, suas características de implementação, seus benefícios e problemas.

Capítulo 4 apresenta o arcabouço proposto por essa dissertação. Nesse capítulo é mostrada a arquitetura do arcabouço, e como é feita a implementação de um compilador usando o arcabouço, onde destacam-se a construção dos analisadores léxico, sintático, semântico, a geração de código e a classe dos nodos da *Árvore Sintática Abstrata* (*Abstract Syntax Tree* AST).

Capítulo 5 mostra a avaliação do arcabouço em relação à facilidade de uso e recursos para alteração da implementação dos compiladores. Esse capítulo também mostra a validação do arcabouço, destacando a implementação do compilador de Machĭna.

Capitulo 6 mostra as contribuições deste trabalho e os possíveis trabalhos futuros.

Apêndice A apresenta a gramática de especificação do Gerador de Front-End. Apêndice B mostra a implementação de um compilador para uma pequena linguagem de programação.

Capítulo 2

Revisão da Literatura

Neste capítulo, são apresentados os conceitos de arcabouços, padrões de projeto com ênfase no padrão Visitor, e a programação orientada por aspectos. Esses conceitos são os mais importantes dentro do processo de desenvolvimento do arcabouço proposto nesta dissertação.

2.1 Arcabouços

Ralph Johnson fornece duas definições para arcabouços[1] [42]. A primeira define que arcabouço é um conjunto de classes abstratas e uma representação da maneira pela qual tais classes se interagem. A segunda define arcabouços como o esqueleto de uma aplicação que pode ser customizado de acordo com as intenções de um desenvolvedor de aplicações. Segundo Johnson, essas duas definições não são conflitantes, pois a primeira descreve a estrutura de um arcabouço, enquanto a segunda descreve o seu propósito.

Arcabouço é uma técnica de reúso. A reusabilidade é definida como a propriedade de um software ser reusável em novas aplicações. Uma das formas de reúso é o reúso de código por meio de componentes. Componente é um conjunto de classes ou de objetos que estão intimamente relacionados e que oferece uma funcionalidade específica. Os componentes podem ser facilmente conectados para a criação de novos sistemas. Uma outra forma de reúso é o reúso de projeto que consiste na utilização de padrões de projeto, descritos na Seção 2.2. Arcabouços permitem tanto o reúso de código quanto o reúso de projeto.

Um arcabouço é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma específica classe de software. A customização de um arcabouço para uma dada aplicação pode ser feita com a criação de

[1] do inglês frameworks

subclasses específicas para a aplicação, as quais são subclasses das classes abstratas do arcabouço. Outra maneira de especializar é via o uso de componentes que acompanham o arcabouço. Esses componentes são implementações das classes abstratas do arcabouço que permitem ao usuário escolhas entre os componentes para construir sua aplicação. Portanto o reúso de código em um arcabouço pode ser feito pelo uso dos componentes existentes ou por meio da herança, aproveitando boa parte da implementação da superclasse.

O arcabouço determina a arquitetura da aplicação. Ele define a estrutura geral, sua divisão em classes e objetos e como conseqüência, as responsabilidades das classes de objetos, como as mesmas colaboram entre si, e o fluxo de controle. Um arcabouço predefine esses parâmetros de projeto, liberando o implementador da aplicação para se concentrar nos aspectos específicos de sua aplicação, promovendo desse modo reúso de projeto.

Uma das mais importantes características de um arcabouço é a *inversão de controle*. Tradicionalmente, um desenvolvedor de software reúsa componentes de uma biblioteca em seu programa. Ele decide quando chamar os componentes e são de sua responsabilidade a estrutura geral e o fluxo de controle do programa. Em arcabouços, é o programa principal que é reusado pelo desenvolvedor da aplicação, que por sua vez decide que componentes incluir a esse programa principal e também pode criar novos componentes. Desse modo, o código do desenvolvedor da aplicação é chamado pelo código do arcabouço, caracterizando assim a *inversão de controle*.

Arcabouços estão no meio das técnicas de reúso. Eles são mais abstratos e flexíveis, porém mais difíceis de serem usados que os componentes. Por outro lado, eles são mais concretos e fáceis de serem reutilizáveis, porém menos flexíveis e aplicáveis que os padrões de projetos.

2.1.1 Arcabouços Caixa Branca e Caixa Preta

Arcabouços podem ser classificados em sistemas do tipo *caixa branca* e sistema do tipo *caixa preta* [28, 42].

Arcabouços *caixa branca* utilizam características das linguagens orientadas por objetos como herança, amarração dinâmica, redefinição de métodos como técnica de extensão. Arcabouços *caixa preta* suportam extensibilidade por definição de interfaces para componentes que podem ser plugados no arcabouço via composição de objetos.

Arcabouços *caixa branca* demandam que o desenvolvedor da aplicação conheça em detalhe a estrutura interna do arcabouço. Em contraste, os arcabouços *caixa preta* são estruturas que usam composição de objetos em vez de herança. Como resultado, o desenvolvedor não precisa ter um profundo conhecimento sobre a estrutura interna do arcabouço. Este fato os faz mais

fáceis de serem usados, porém com uma menor flexibilidade em relação aos arcabouços *caixa branca*. Os arcabouços *caixa preta* também são mais difíceis de serem desenvolvidos, pois requerem que o desenvolvedor do arcabouço defina uma interface e os componentes que antecipem os potenciais casos de uso.

2.1.2 Benefícios e Problemas

Os principais benefícios dos arcabouços são obtidos pelo reúso de código e de projeto. Alguns desses benefícios são:

- evita recriar e reavaliar soluções antes criadas.
- Maior produtividade para os desenvolvedores da aplicação, que podem dedicar às características específicas de sua aplicação.
- Introduz maior qualidade e confiabilidade nas aplicações.
- Reduz a manutenção.
- Reparos feitos no arcabouço propagam-se nas aplicações.
- Requer menor tempo de desenvolvimento em novas aplicações.

Arcabouços apresentam problemas devido à sua complexidade. Alguns desses problemas são:

- aprender a usar um arcabouço é difícil, o que obriga ter um tempo considerável de treinamento.
- O desenvolvimento de um arcabouço com qualidade, flexibilidade, extensibilidade, reusabilidade para aplicações complexas é uma tarefa difícil.
- As estruturas genéricas que melhoram a flexibilidade e a extensibilidade do arcabouço, dificultam a depuração de código, pois essas estruturas não podem ser depuradas separadamente de suas específicas instanciações. Devido a isso, é difícil distinguir erros de código do arcabouço dos erros de código da aplicação.
- A inversão de controle também torna difícil o processo de depuração.
- Arcabouços são limitados a apenas uma linguagem orientada por objetos. Diferentes linguagens orientadas por objetos não trabalham bem juntas.

2.1.3 Conclusão

O arcabouço implementado nesta dissertação é do tipo *caixa branca*. O usuário fica responsável pela redefinição de métodos abstratos das classes da *Árvore Sintática Abstrata*(AST), enquanto o arcabouço fica responsável pela arquitetura do compilador e do fluxo de controle. Portanto, esses parâmetros de projeto são aproveitados em todos os compiladores desenvolvidos, sendo promovido deste modo o reúso de projeto. O reúso de código é obtido por meio do reaproveitamento de classes pré-definidas ou geradas automaticamente.

O arcabouço do tipo *caixa branca* obriga que o usuário tenha conhecimentos mais detalhados sobre o sistema. Porém, o arcabouço foi projetado para ser de fácil utilização. A maioria das informações é conhecida pelo usuário em sua própria definição de entrada para o gerador de front-end, por exemplo, classes dos nodos da AST, nomes de membros e métodos destas classes, passos de compilação, métodos abstratos que terão que ser redefinidos, entre outras informações. Portanto, o usuário tem que conhecer a linguagem de especificação para o gerador de front-end e o que é gerado a partir desta especificação para que possa utilizar o arcabouço.

O arcabouço permite também que seja gerado um compilador usando apenas as especificações para o gerador de front-end. Este compilador gerado possui apenas as análises léxica e sintática, permitindo deste modo que o usuário faça testes e depurações nestas fases antes de qualquer código para a análise semântica ou geração de código sejam implementados. Isto permite que sejam totalmente separados os testes e depurações do código gerado ou pré-definido pelo arcabouço do código implementado pelo usuário.

2.2 Padrões de Projeto

Padrões de projeto têm recentemente se tornado uma forma de reúso de projeto. Um padrão descreve o problema a ser resolvido, a solução, o contexto na qual a solução trabalha, seus custos e benefícios [42].

Padrões de projeto capturam soluções que foram desenvolvidas e aperfeiçoadas ao longo do tempo. Eles refletem modelagens resultantes dos esforços dos desenvolvedores por maior reutilização e flexibilidade em seus sistemas.

A mais importante referência sobre padrões de projeto é o livro do *Gamma* et alii [30]. Esse livro apresenta um catálogo com 23 padrões.

A utilização de padrões de projeto no desenvolvimento deste trabalho de dissertação se justifica em primeiro lugar porque eles facilitam o projeto de

sistema de softwares complexos e em segundo lugar porque tais padrões, em especial o padrão Visitor, têm sido muito usados pelos compiladores modernos. Dada a sua importância no desenvolvimento deste projeto, o padrão Visitor é descrito a seguir.

2.2.1 Padrão Visitor

O padrão Visitor permite adicionar um ou mais comportamento em uma classe dentro de uma hierarquia, enquanto define esses novos comportamentos em outra hierarquia de classes.

A Figura 2.1 mostra um exemplo da estrutura desse padrão. Cada uma das classes *ConcreteVisitor1* e *ConcreteVisitor2* descreve um novo comportamento cuja intenção é adicioná-las em cada subclasse da classe abstrata *Element*.

Objetos da subclasses de *Element*, em algum momento da execução, são visitadas com um determinado objetivo. Esse objetivo pode ser a execução de um dos comportamentos definidos nas classes do Visitor. Para que esses novos comportamentos sejam invocados, o método *accept* é chamado. O método *accept* recebe como parâmetro um objeto do tipo Visitor. Observe que esse parâmetro pode receber qualquer um dos objetos Visitor da hierarquia e, a partir desse objeto Visitor, o novo comportamento é chamado pelo método *visitConcreteElementA* se o *accept* pertencer à classe *ConcreteElementA* ou pelo método *visitConcreteElementB* se o *accept* pertencer à classe *ConcretElementB*, como é visto no exemplo.

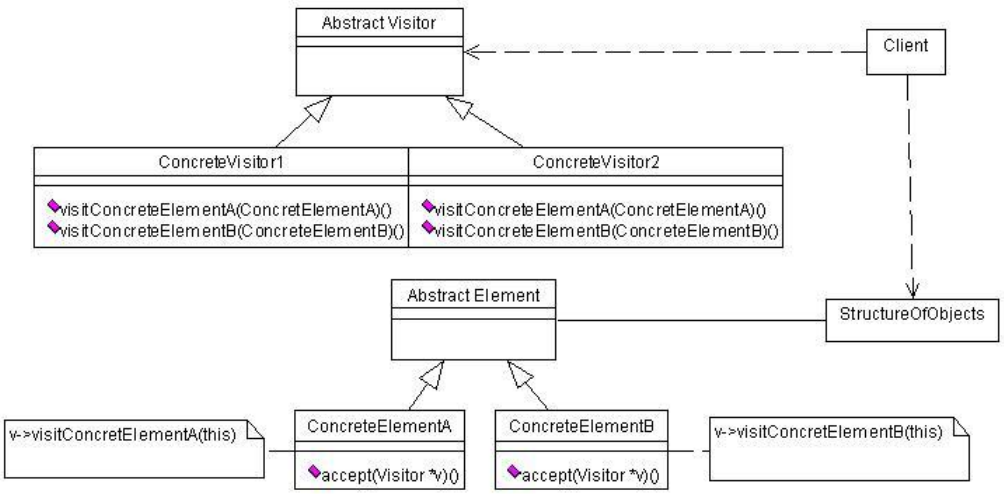

Figura 2.1: Padrão Visitor

O principal benefício no uso do padrão Visitor está relacionado ao fato do mesmo poder adicionar novos comportamentos a uma hierarquia de classes sem modificá-las, apenas criando novas subclasses do Visitor. Embora muito úteis, cuidados devem ser tomados no uso do padrão Visitor e de outros padrões de projetos devido aos seguintes problemas [38]:

Problema da Confusão: o uso de padrões torna difícil distinguir entre as definições resultantes 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 um comportamento de uma hierarquia de classes definida nas classes do Visitor. Isso faz com que o programa seja difícil de ser entendido, aumentando a comunicação entre objetos, além de introduzir dependências que podem impedir 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, os métodos *visitConcreteElementA* e *visitConcreteElementB* das classes *ConcreteVisitor1* e *ConcreteVisitor2* mostrados na Figura 2.1 podem necessitar usar os atributos das classes *ConcreteElementA* e *ConcreteElementB* respectivamente. Para que o padrão Visitor seja eficiente, esses atributos precisam ser declarados como públicos, ou pelo menos ter algum 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 forem consideradas algumas variações onde os novos comportamentos são adicionadas a classes de diferentes hierarquias esse problema pode ocorrer.

2.2.2 Conclusão

O padrão de projeto Visitor tornou-se popular na construção de compiladores devido sua característica descrita na Seção 2.2.1. Com o padrão Visitor é

possível construir um compilador utilizando classes de nodos de uma AST que tenham sido anteriormente construídas, seja de modo automático ou manual.

Essas classes de nodos da AST são instanciadas na análise sintática do compilador, de modo a resultar na AST que é uma representação do programa contido no arquivo fonte. O compilador pode depois desta etapa usar a AST para a realização de análise semântica e geração de código. Para isso, o compilador percorre a AST visitando cada um dos seus nodos. Portanto, essas classes da AST não precisam possuir esses comportamentos de análise semântica e geração de código no momento de sua criação, podendo deixar que o Visitor adicione esses novos comportamentos posteriormente, o que permite que essas classes sejam geradas automaticamente.

Contudo, o padrão de projeto Visitor apresenta problemas como os mostrados na Seção 2.2.1. Esses problemas podem dificultar modificações nos compiladores implementados, dificultando sua evolução.

Devido à necessidade do arcabouço implementado nesta dissertação de proporcionar extensibilidade, flexibilidade e modularização para que futuras modificações nos compiladores possam reusar o máximo da implementação anterior, o uso do padrão de projetos Visitor no arcabouço foi descartado e substituído pela programação orientada por aspectos.

2.3 Programação Orientada por Aspectos

A programação orientada por aspectos foi proposta por Gregor Kiczales [45] com o objetivo de modularizar decisões de projeto que não podem ser adequadamente implementadas por meio da programação orientada por objetos.

Na programação orientada por aspectos, os requisitos de sistemas são modularizados por meio de classes e aspectos. Os aspectos implementam requisitos transversais do sistema, tais como geração de registro de operações, tratamento de erros, segurança, comunicação.

Os mecanismos de programação orientados por aspectos são também uma poderosa ferramenta de implementação de código, permitindo alterar a estrutura interna das classes, como na implementação de padrões de projeto.

Existem várias linguagens orientadas por aspectos, entre elas destacam-se as implementações para Java: *AspectJ* [44, 48, 5], *HyperJ* [58]; a implementação para C++: *AspectC++* [4, 64]; a implementação para C: *AspectC* [23].

Dentre as aplicações desenvolvidas usando programação orientada por aspectos, destacam-se a reestruturação do núcleo do sistema operacional *FreeBSD* [22, 23], a implementação de sistemas distribuídos [60, 63], a imple-

mentação de arcabouços [24] e a implementação de persistência como requisito transversal [61, 63].

2.3.1 Requisitos de Sistemas

Um módulo é um artefato de programação que pode ser desenvolvido separadamente de outras partes que compõem o programa. Na programação estruturada, a modularização se limita à implementação de abstrações dos comandos necessários à realização de uma tarefa (procedimentos e funções). Na programação orientada por objetos, pode-se obter um grau mais elevado de modularização, é possível realizar abstrações de estruturas de dados, tipos e de suas operações [70].

Desenvolvedores criam sistemas a partir de requisitos que podem ser classificados como:

- *Requisitos funcionais*: que constituem o objetivo do sistema.
- *Requisitos não-funcionais*: que compreendem elementos específicos de projeto, muitas vezes sem relação direta com o problema em questão. Por exemplo, registro de operações (*logging*), garantia de integridade de transações, segurança e desempenho.

A orientação por objetos permite uma boa modularização dos requisitos funcionais do sistema, mas não é capaz de resolver adequadamente problemas relacionados à modularização de requisitos não mapeáveis diretamente em uma ou poucas classes de um sistema, tendendo a se espalhar por todo código de programa [70]. Estes requisitos são denominados *requisitos transversais*.

2.3.2 Requisitos Transversais

Requisitos transversais são também denominados aspectos. Em sistemas orientados por objetos, a unidade de modularização é a classe, e os requisitos transversais se espalham por múltiplas classes. A falta de tratamento adequado aos requisitos transversais pode resultar no baixo grau de modularização do sistema. Por estar espalhado em diversos módulos do sistema, torna-se difícil entender, implementar e modificar códigos relacionados à implementação de um desses requisitos.

Esses problemas são exemplificados na classe definida na Figura 2.2 extraída de [70]. Nesse código, alguns problemas podem ser identificados:

- A parte de *Outros dados membros* trata dos dados relativos a logging, autenticação, consistência etc, que não pertence ao objetivo principal da classe.

```
class Something {
    "Dados membros do módulo"
    "Outros dados membros"

    "Métodos redefinidos da superclasse"

    public void performSomeOperation(OperationInformation info)
    {
        "Garante autenticidade"
        "Bloqueia o objeto para garantir consistência
         caso outras threads o acessem"
        "Garante que a cache está atualizada"
        "Faz o logging do início da operação"
        "REALIZA A OPERAÇÃO OBJETIVO"
        "Faz o logging do final da operação"
        "Desbloqueia o objeto"
    }

    "Outras operações semelhantes à anterior"

    public void save(PersistenceStorage ps) {...}
    public void load(PersistenceStorage ps) {...}
}
```

Figura 2.2: Requisitos Transversais na POO

- O método *performSomeOperation* faz mais operações de requisitos transversais do que realiza a *operação objetivo*.
- Não é claro se as operações *save* e *load*, que realizam a gerência de persistência, são métodos da parte principal da classe.

Outro problema é que modificações em algum dos requisitos transversais podem originar modificações em diversas partes do programa. Os problemas da modularização de requisitos transversais podem ser divididos em *espalhamento* e *intrusão*:

- Espalhamento diz respeito a código para implementação de um requisito transversal que esteja disperso ao longo de todo programa.
- Intrusão diz respeito à confusão gerada por código de mais de um requisito transversal estar presente em uma única região do programa, como no método *performSomeOperation*.

As implicações desses problemas são diversas:

- Dificultam o rastreamento do programa: vários requisitos em um único método tornam difícil a correspondência entre os requisitos e sua implementação.
- Baixa produtividade: a implementação simultânea de vários requisitos em um módulo pode desviar a atenção do desenvolvedor do requisito principal.
- Baixo grau de reúso: dado que um único módulo implementa vários requisitos, outros sistemas que precisam implementar funcionalidade semelhante podem não ser capazes de utilizar prontamente o módulo.
- Pouca qualidade interna: ao se preocupar com diversos requisitos ao mesmo tempo, o desenvolvedor pode não dar atenção suficiente a um ou mais requisitos.

Todos estes problemas levam à produção de código de difícil evolução ou baixa extensibilidade, visto que modificações posteriores no sistema podem ser dificultadas pela pouca modularização.

2.3.3 Desenvolvimento de Software Orientado por Aspectos

A programação orientada por aspectos tem como objetivo permitir a definição separada de requisitos transversais às classes de um sistema orientado por objeto.

Os requisitos transversais são implementados em módulos chamados aspectos. Os aspectos passam pelo processo de costura de código (*weaving*) realizado pelo compilador de aspectos, entrelaçando os códigos dos aspectos com os códigos das classes.

Existem dois tipos de implementação dos requisitos transversais: a *transversalidade dinâmica* e a *transversalidade estática*.

Linguagens orientadas por aspectos devem possuir como elementos básicos para uso da transversalidade dinâmica um modo de definição de *pontos de junção*. Pontos de junção são posições bem definidas de execução de um programa, por exemplo, chamadas de métodos ou execução de blocos de tratamentos de exceção. Associados aos pontos de junção existem mais dois conceitos importantes: *conjunto de junção* e *regras de junção*.

Um conjunto de junção é formado por um conjunto de pontos de junção identificados por um padrão contextual e sintático e tem a função de reunir

informações a respeito de contexto destes pontos. Conjuntos de junção podem ser visto como registros de identificação de pontos de junção em tempo de execução.

Regras de junção representam os códigos que devem ser executados em pontos de junção.

Algumas implementações de aspectos necessitam de recursos para alterar tanto o comportamento das classes em tempo de execução quanto a sua estrutura estática.

A transversalidade dinâmica obtida a partir de regras de junção permite modificar o comportamento da execução do programa, ao passo que a transversalidade estática permite redefinir a estrutura estática dos tipos – classes, interfaces ou outros aspectos – e o seu comportamento em tempo de execução.

Em AspectJ, é possível implementar os seguintes tipos de transversalidade estática:

- Introdução de membros e métodos em classes e interfaces.
- Modificação das hierarquias de tipos.
- Declaração de erros e advertências de compilação.
- Enfraquecimento de exceções.

Em AspectC++, são os seguintes tipos de transversalidade estática:

- Introdução de membros, métodos, construtores e destrutores em classes.
- Introdução de tipos.
- Introdução de funções.
- Introdução de uma nova classe base em uma classe.

2.3.4 Conclusão

O desenvolvimento orientado por aspectos permite definir claramente as responsabilidades dos módulos individuais, uma vez que cada módulo é responsável unicamente por seu requisito principal. Além disso, o nível de modularização do sistema é melhorado, com baixo acoplamento e alta coesão modular. Com efeito, ao retirar código intruso dos módulos, é possível diminuir a interface de cada módulo e, ao mesmo tempo, aumentar o seu nível de coesão, pois trata somente um requisito. As conseqüências diretas desses

fatos são a melhoria do processo de manutenção de sistemas e aumento no seu grau de reúso.

Além disso, a evolução de sistemas é facilitada, haja vista que, se novos aspectos forem criados para implementar novos requisitos transversais, as classes do programa podem permanecer inalteradas. Da mesma forma, ao adicionar novas classes ao programa, os aspectos existentes também são transversais a essas classes.

A programação orientada por aspectos não é um substituto para a programação orientada por objetos. Os requisitos funcionais de um sistema continuarão a ser implementados por meio da programação orientada por objetos. A orientação por aspectos simplesmente adiciona novos conceitos à orientação por objetos, facilitando a implementação de requisitos transversais e retirando grande parte da atenção dada a tais requisitos nas fases iniciais de desenvolvimento.

A programação orientada por aspectos é usada no arcabouço para a implementação da análise semântica e geração de código substituindo o padrão Visitor usado para essa funcionalidade em outros arcabouços e compiladores. Seu uso se dá por meio do uso da inserção estática de membros e métodos em classes da transversalidade estática. Como é mostrado no Capítulo 4, a inserção estática não apresenta os problemas encontrados no padrão Visitor, possibilitando portanto que a extensibilidade do arcabouço seja de fácil utilização e entendimento, além de permitir maior independência entre as classes, o que possibilita uma maior flexibilidade para modificações nos compiladores desenvolvidos.

2.4 Conclusão

Este capítulo apresentou o conceito de arcabouço, mostrando seu poder e flexibilidade obtida pelo reúso de código e de projeto. Ainda em relação ao reúso de projeto discutiu-se sobre os padrões de projeto. Outra técnica descrita foi a programação orientada por aspectos, que oferece maior poder de modularização, permitindo implementar requisitos transversais separadamente dos requisitos funcionais. Essas técnicas ajudam a implementar softwares complexos, permitindo uma maior modularização, reusabilidade, flexibilidade, etc.

Atenção especial foi dada ao padrão Visitor usado por muitos compiladores modernos para a implementação de análise semântica e geração de código.

A utilização desses conceitos em uma ferramenta de compilação é mostrada no Capítulo 3 com os arcabouços: *Polyglot*, *JastAdd* e *SableCC*.

O Capítulo 4 apresenta a ferramenta proposta por essa dissertação. Esta ferramenta obtém o reúso de código e de projeto com o uso da técnica de arcabouço e a utilização da programação orientada por aspectos para a implementação de análise semântica e geração de código. A programação orientada por aspectos foi escolhida em vez do padrão de projeto Visitor devido aos problemas deste padrão mostrados na Seção 2.2.1. Ainda no Capítulo 4, é descrito como isso é feito e quais as vantagens.

Capítulo 3

Ferramentas de Compilação

Este capítulo apresenta três arcabouços de compilação: *Polyglot*, *JastAdd* e *SableCC*. Para cada arcabouço é destacada sua arquitetura, seu modo de uso, suas características para a implementação dos analisadores léxico, sintático e semântico. No final de cada descrição são discutidas as vantagens e as desvantagens de cada arcabouço.

3.1 Arcabouço Polyglot

O *Polyglot* [55] é um arcabouço de compilação que aceita como entrada um código escrito em uma linguagem Java estendida e o transforma para código Java puro. O código Java gerado pode então ser compilado por um compilador Java, por exemplo, o *javac* [66] e ser transformado em *bytecodes*.

O arcabouço *Polyglot*, além de facilitar a criação de compiladores para linguagens similares à Java, é usado para especificação de linguagens, exploração de projeto de linguagens, e para implementações de versões simplificadas de Java direcionadas aos estudantes.

O arcabouço, escrito em Java, é por *default* um simples verificador semântico para a linguagem base Java. O *Polyglot* possui já implementado: o analisador léxico e sintático para a linguagem Java, um conjunto de classes de nodos da AST que representam comandos, expressões, declarações, etc, da linguagem Java que são usadas para a criação da AST em várias etapas do processo de compilação e passos de análise semântica e geração de código. O desenvolvedor de uma linguagem que estende Java pode especializar o arcabouço para definir qualquer mudança necessária para o processo de compilação, seja alterando a sintaxe de Java o que vai necessitar a implementação de novas classes da AST, ou seja alterando a semântica de Java, o que vai necessitar implementar novos passos de compilação.

Um dos importantes objetivos do *Polyglot* está relacionado à extensibilidade escalável: uma extensão deve requerer esforços de programação proporcionais à diferença entre a linguagem estendida e a linguagem base. Adicionar novas classes de nodos da AST ou novos passos de compilação deve requerer escrita de código do tamanho proporcional às mudanças. Normalmente, a extensibilidade escalável é difícil de ser obtida para extensões simultâneas dos nodos da AST e dos passos de compilação. O *Polyglot* propõe uma metodologia para extensibilidade escalável que suporta ambas as extensões.

3.1.1 Arquitetura do Polyglot

A arquitetura do *Polyglot*, extraída de [55], é exibida na Figura 3.1.

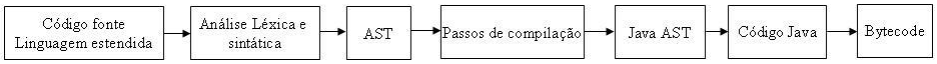

Figura 3.1: Arquitetura do Polyglot

O primeiro passo da compilação é a análise léxica e a análise sintática que produzem a AST através da instanciações das classes de nodos da AST utilizando fábrica de nodos. O Polyglot usa as ferramentas JFlex [46] para gerar o analisador léxico e o PPG [17] para gerar o analisador sintático.

A fábrica de nodos da AST é uma classe que cria os nodos da AST. Cada método da classe cria um nodo diferente. Essa classe pode ser especializada pelo desenvolvedor para que novos nodos da linguagem estendida possam ser criados.

O processo de compilação continua com uma série de passos de compilação aplicados à AST. O escalonador de passos seleciona esses passos na ordem definida pelas informações da linguagem estendida. Cada passo, obtendo sucesso, reescreve a AST, produzindo uma nova AST, que constitui a entrada para o próximo passo. Alguns passos podem apenas relatar erros sem reescrever a AST. Uma linguagem estendida pode modificar o escalonador de passos da linguagem base, adicionando, substituindo ou removendo os passos.

No fim de todos os passos é gerada a AST contendo apenas nodos da base de Java, ou seja, não possui nenhum nodo da extensão de Java. Posteriormente, essa AST é transformada em código Java.

A vantagem de usar o gerador de analisadores sintático PPG é que o mesmo permite implementar definições sintáticas da linguagem estendida como um conjunto de mudanças da gramática da linguagem base. Ele provê herança de gramática, na qual é possível adicionar, modificar e renomear

produções e símbolos da gramática base. O PPG é implementado como um pré-processador para CUP [1].

Para a utilização do *Polyglot*, inicialmente o usuário implementa uma classe de configuração. Essa classe determina qual fábrica de nodos da AST é usada, quais são os passos de compilação, a ordem dos passos, extensão dos arquivos fontes do compilador, entre outras informações. Em seguida, o usuário tem que implementar as alterações necessárias para a geração do compilador para a linguagem estendida.

Essas alterações são feitas usando os mecanismos de extensibilidade propostas para o *Polyglot* que são mostrados na Seção 3.1.3. Cada mecanismo é adequado para um determinado tipo de alteração, por exemplo, para adição de novas classes da AST utiliza-se herança, enquanto para adição de novos passos de compilação utiliza-se delegação.

O usuário pode fazer alterações em questões funcionais do compilador, por exemplo, alterações léxicas, sintáticas, semânticas e de geração de código. Requisitos de projeto, como arquitetura, hierarquias de classes e fluxo de controle ficam sobre responsabilidade do *Polyglot*.

3.1.2 Exemplo

Esta seção apresenta um exemplo de uma linguagem simples denominada *Coffer* [55], que estende Java tanto do ponto de vista sintático como semântico. Esse mesmo exemplo é utilizado na Seção 3.1.3 para ilustrar a extensibilidade do sistema *Polyglot*.

Coffer faz com que os objetos sejam associados a uma chave. Métodos de um objeto só podem ser invocados quando a chave é segura. A chave é alocada quando o objeto é criado e liberada pela declaração *free*.

A Figura 3.2 mostra um exemplo de *Coffer*. Nas linhas de 2 a 9 é declarada a classe *FileReader*, garantindo que um arquivo só possa ser lido a partir de um leitor que tenha aberto o arquivo. Na linha 2, a anotação *tracked(F)* associa a chave de nome F com a instância de *FileReader*. Pré e pós-condições dos métodos e construtores são escritos entre colchetes nas suas assinaturas. Por exemplo, o construtor na linha 3 tem como pré-condição [], indicando que não é necessária nenhuma chave para acessá-lo. Sua pós-condição é definida por [F] e especifica que F é a chave segura quando o retorno do construtor é normal. No método *close* a chave F é liberada, logo nenhum método que requer F como pré-condição pode ser acessado. A classe *SimpleTest*, da linha de 11 a 18, mostra a utilização da classe *FileReader*.

Coffer estende Java do seguinte modo: adiciona a declaração *free*, altera a sintaxe de métodos e construtores e adiciona novos passos como, por exemplo, *KeyFlow* e *checkKeys*.

```
1
2    tracked(F) class FileReader {
3      public FileReader() [] -> [F] {...}
4      public int read() [F] -> [F] {...}
5      public void close() [F] -> [] {
6          ... ;
7          free this;
8      }
9    }
10
11   public class SimpleTest {
12     public static void main(String[] args) {
13         tracked(I) FileReader is =
14                 new tracked(I) FileReader();
15         is.read();
16         is.close();
17     }
18   }
```

Figura 3.2: Exemplo do Coffer

3.1.3 Extensibilidade

Polyglot propõe um mecanismo de extensibilidade escalável para a sintaxe e a semântica da linguagem base com o objetivo de não duplicar o código.

Existem outras formas de estender a linguagem base, por exemplo, o padrão Visitor (Seção 2.2.1), muito embora seja uma metodologia de extensibilidade não escalável, permite adicionar novos passos de modo escalável, mas sacrifica a adição de novas classes de nodos da AST. Para cada novo nodo adicionado é necessário modificar todas as classes do Visitor existentes, adicionando um novo método.

Outra metodologia de extensibilidade não escalável é usar os passos de compilação como métodos dos nodos da AST. Essa metodologia permite adicionar novas classes de nodos da AST de modo escalável, mas não consegue o mesmo para adição de novos passos de compilação. Para cada novo passo adicionado, é necessário adicionar um novo método para cada nodo da AST.

A metodologia adotada pelo *Polyglot* consiste na implementação de cada passo de compilação como métodos dos nodos da AST introduzindo mecanismos de delegação para que seja possível a extensibilidade escalável.

A adição de novos nodos da AST é feita via herança. Por exemplo, a linguagem *Coffer* adiciona o nodo *Free* na hierarquia de nodos usando herança como mostrado na Figura 3.3[1]. O mecanismo de herança também é

[1]Nenhuma das Figuras dessa seção tem por objetivo mostrar toda a complexidade das

usado em outras partes extensíveis do *Polyglot*, como por exemplo, na fábrica de nodos e no escalonador de passos.

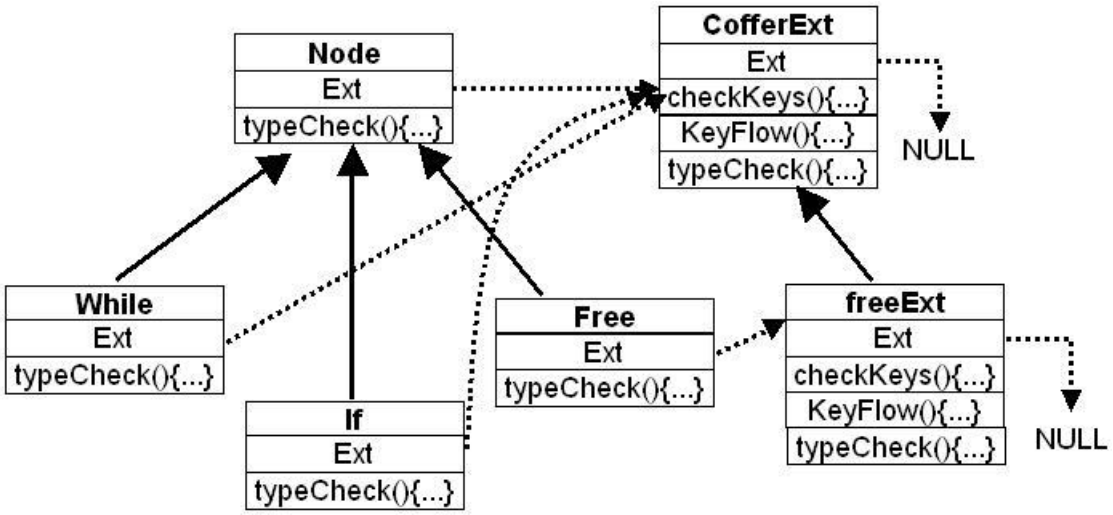

Figura 3.3: Objeto de Extensão

Ao criar novos passos, é necessário estender as classes dos nodos da AST sem alterá-las e sem usar herança, para que seja mantida a extensibilidade escalável e evitar a duplicação de código. Para que isso seja possível, o *Polyglot* usa um mecanismo de delegação, todos os nodos da AST possuem um ponteiro chamado *Ext*. Esse ponteiro, quando diferente de *null*, aponta para um *objeto de extensão* que adiciona novos membros e métodos nas classes da AST. A Figura 3.3[2] mostra o *objeto de extensão CofferExt* que estende *Node* com a adição dos métodos *KeyFlow()* e *checkKeys()*. Cada nodo da AST pode ser estendido com uma implementação específica, como pode ser observado na Figura 3.3, para o nodo *Free*. Outros nodos usam o *objeto de extensão* com uma implementação *default*, evitando desse modo a duplicação de código, como pode ser visto para os nodos *If* e *While*. Os *objetos de extensão* também possuem o ponteiro *Ext*, por isto, a linguagem de extensão também pode ser estendida.

Entretanto, o *objeto de extensão* não é adequado para sobrescrever métodos. O problema é que qualquer *objeto de extensão* pode sobrescrever quaisquer métodos, como foi mostrado na Figura 3.3, para o método *typeCheck()*, logo as classes da AST podem usar mais de uma implementação para o mesmo

hierarquias de classes usadas no *Polyglot*, sendo usadas apenas para ilustrar de modo simplificado seu mecanismo de extensão.

[2]As setas pontilhadas apontam para o objeto onde os ponteiros *Exp* e *Del* apontam. As outras setas significam herança.

método. Para resolver esse problema, o *Polyglot* usa novamente o mecanismo de delegação.

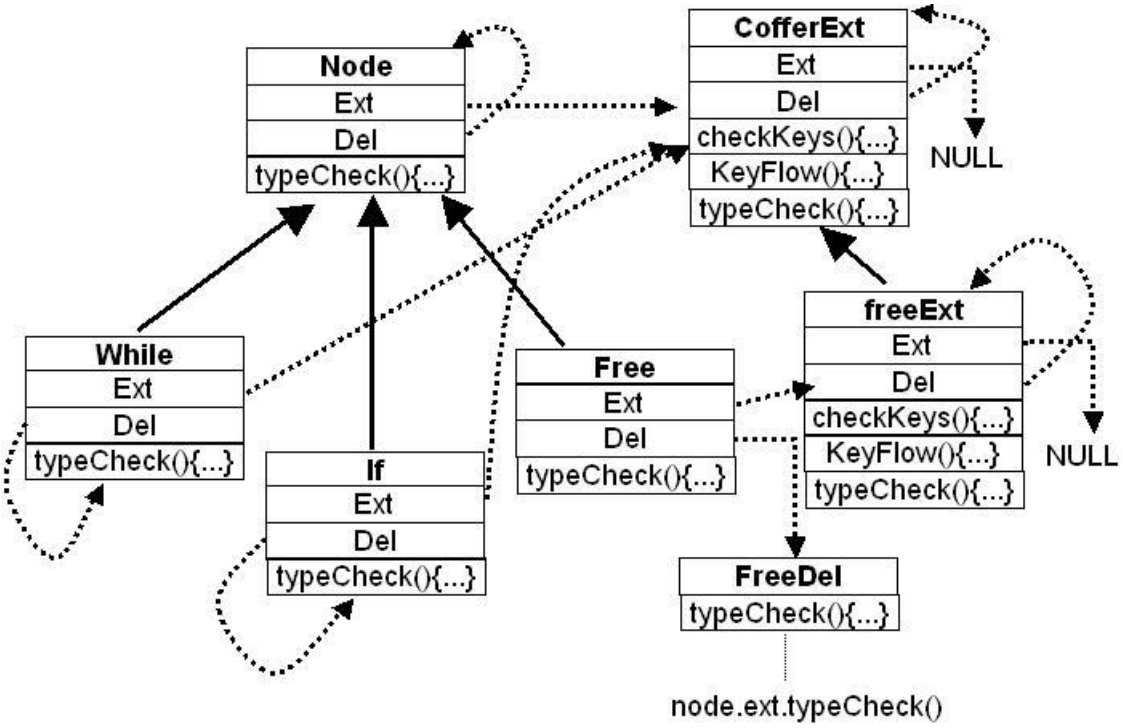

Figura 3.4: Objeto de Delegação

Todas as classes da AST têm um ponteiro chamado *Del*, que aponta para um *objeto de delegação*. O *objeto de delegação* implementa a interface de *Node* e por *default* o ponteiro *Del* aponta para o próprio nodo. O *objeto de delegação* pode possuir uma nova implementação do método, ou invocar a implementação correta. A Figura 3.4 mostra a solução para o método *typeCheck()* da Figura 3.3, onde o nodo *Free* usa o *typeCheck()* definido no *objeto de extensão FreeExt* e o restante usa a implementação da sua própria classe. *Objetos de extensão* também contêm o ponteiro *Del*, que aponta para um *objeto de delegação* que implementa a interface do *objeto de extensão*. Desse modo, é possível também sobrescrever seus métodos.

A chamada de todos os métodos é feita pelo ponteiro *Del*, garantindo a chamada da correta implementação. No exemplo da Figura 3.4, o método *typeCheck()* é invocado via *n.Del.typeCheck()*, e o método *checkKeys()* é invocado via *((CofferExt)n.Ext).Del.checkKeys()*.

3.1.4 Conclusão

Polyglot é um arcabouço cujo objetivo é gerar compiladores para linguagens que estendem ou simplificam a linguagem Java.

As vantagens do *Polyglot* são a extensibilidade escalável e a não duplicação de código, permitindo que o usuário tenha o menor esforço possível no desenvolvimento do compilador para sua linguagem. O *Polyglot*, portanto, é uma boa ferramenta para projetos de linguagens que estendem Java, permitindo que o usuário faça constantes modificações. Contudo, seus maiores benefícios são também um de seus piores problemas. A extensibilidade escalável e a não duplicação de código faz com que a estrutura de código do *Polyglot* e o seu mecanismo de extensibilidade sejam complexos e difíceis de serem entendidos. 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 na Seção 2.2.1.

Outro problema em relação à utilização do *Polyglot* é a necessidade do usuário ter conhecimento sobre toda a estrutura do código e do fluxo de controle para que possa usá-lo. Obter esses conhecimentos não é uma tarefa simples, pois o *Polyglot* possui muitas classes, e entre elas muitas são abstratas. O uso de delegação é outro fator e, além disso, sua documentação é ineficiente.

Do ponto de vista de seu objetivo, *Polyglot* é uma ferramenta poderosa, mas que possui poucas perspectivas de difundido uso nas comunidades de pesquisadores e desenvolvedores devido à sua dificuldade de aprendizagem e de uso.

O *Polyglot* não teve influência neste trabalho, pois seu objetivo é diferente. Este trabalho tem como objetivo criar compiladores para qualquer linguagem, e o objetivo do *Polyglot* é criar compiladores a partir de uma linguagem base, o que o torna praticamente inviável a sua utilização quando a linguagem estendida é muito diferente da linguagem base.

3.2 Arcabouço JastAdd

JastAdd [40] é um arcabouço para especificação e implementação de compiladores que gera automaticamente as classes de nodos da AST, permitindo que as análises semânticas e a geração de código sejam implementadas convenientemente por meio do uso da AST. As análises semânticas e a geração de código são modularizadas em diferentes módulos, que são costurados junto às classes da AST usando técnica de programação orientada por aspectos.

Os módulos de implementação da análise semântica e da geração de código são de dois tipos: módulos imperativos e módulos declarativos. Os módulos imperativos são implementados usando a linguagem Java. Os módulos declarativos usam *Reference Attributed Grammars* (RAGs) [39]. A linguagem que implementa o RAG é uma extensão da linguagem Java, que no processo de desenvolvimento é transformada para código Java pelo sistema.

3.2.1 Arquitetura do JastAdd

A Figura 3.5 mostra a arquitetura do *JastAdd*. Os arquivos *.jadd*, *.jrag*, *.ast* e o *.jjt* são implementados pelo usuário, gerando as classes mostradas nessa mesma figura. O usuário também pode criar outras classes, não mostradas na figura, que podem ser usadas com as classes geradas pelo *JastAdd*. Uma dessas classes deve ser a classe principal do compilador, que manipula os arquivos fontes e invoca os analisadores.

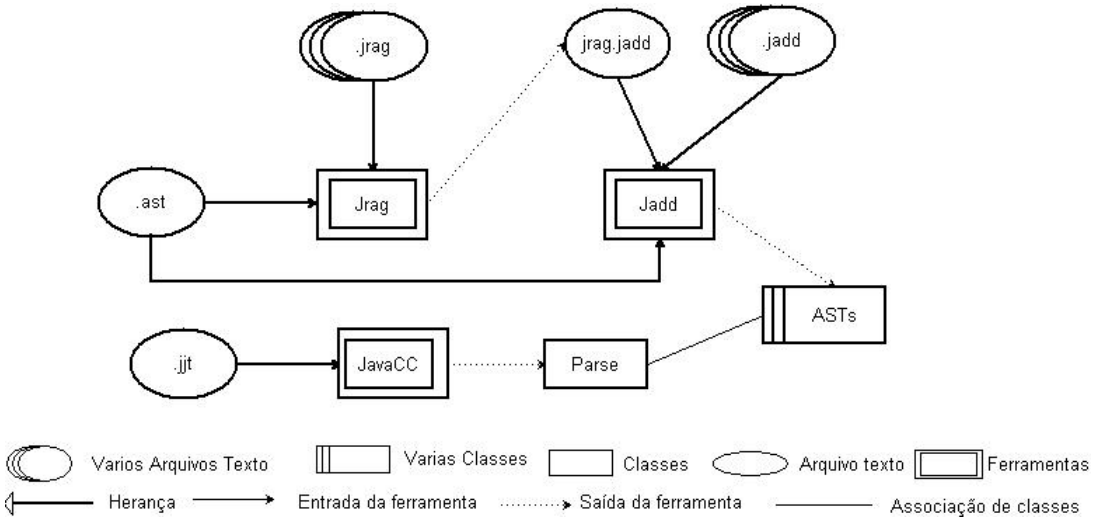

Figura 3.5: Arquitetura do JastAdd

O *JastAdd* possui duas ferramentas: o *Jrag* e o *Jadd*. O *Jadd* gera as classes da AST baseada nas especificações encontradas no arquivo *.ast* e faz as costuras dos módulos imperativos, arquivos *.jadd*, nessas classes. O *Jrag* compila os módulos declarativos, arquivos *.jrag*, gerando um módulo imperativo que é costurado junto com os outros módulos.

3.2.2 Analisador Léxico, Analisador Sintático e Classes da AST

Os analisadores léxico e sintático são gerados pelo *JavaCC* [65], muito embora o *JastAdd* não tenha sua implementação presa ao *JavaCC*. Isso se deve a dois fatores: (i) a especificação da definição das classes de nodos da AST é feita separadamente das definições usadas para a construção dos analisadores léxico e sintático; (ii) o analisador sintático tem apenas como ação semântica a criação da AST via instanciações das classes criadas pelo *Jadd*. Isto posto, o *JastAdd* pode usar qualquer outro gerador de analisadores léxicos e sintáticos.

O *JavaCC* possui uma ferramenta chamada *JJTree* que permite criar classes da AST e facilita seu o processo de criação durante a análise sintática. Entretanto, se o desenvolvedor precisar modificar as classes geradas pelo *JJTree*, é necessário alterar o seu código. Alteração de códigos gerados automaticamente é um processo que pode gerar erros e deve ser evitado. No *JastAdd*, existe a distinção entre código gerado automaticamente e código gerado pelo usuário, desse modo o desenvolvedor nunca tem que modificar os códigos gerado pelo *JastAdd*.

3.2.3 Análise Semântica e Geração de Código

A implementação da análise semântica e da geração de código é feita pela inserção estática da programação orientada por aspectos. A inserção estática não é feita por nenhuma linguagem orientada por aspectos por exemplo, *AspectJ*. Ela é realizada pela ferramenta *Jadd*, que costura os membros e métodos escritos em linguagem Java nas classes da AST.

Existem dois tipos de módulos que implementam os membros e métodos inseridos nas classes da AST: os módulos imperativos e os módulos declarativos.

Os módulos imperativos são implementados em código Java e já estão prontos para serem costurados pelo *Jadd*. Os módulos declarativos são escritos por uma linguagem que estende e modifica a linguagem Java. Essa linguagem tem apenas as seguintes construções: declarações de atributos, implementações de métodos e expressões.

Atributos são declarados da mesma forma que os atributos de Java, exceto que possuem o modificador *syn* ou *inh*. Esses modificadores fazem com que esses atributos tenham uma diferente tradução para o código Java. O modificador *syn* faz com que o atributo seja transformado em um método da classe na qual será inserido. O modificador *inh* faz com que o atributo seja transformado em um método de uma interface comum a todas as classes da AST que tem esse método inserido.

As implementações de métodos são iguais às de Java, exceto que não podem conter efeitos visíveis fora do método. Esses métodos podem usar qualquer comando de Java, desde que a restrição descrita seja respeitada. As equações são iguais às atribuições de Java, onde o lado esquerdo pode ser um atributo que possui o modificador *syn* ou *inh*.

Os módulos declarativos são compilados transformando-os em código Java pela ferramenta *Jrag* e gerando um módulo imperativo que, junto com os outros módulos imperativos, são costurados nas classes da AST pelo *Jadd*.

3.2.4 Conclusão

O *JastAdd* é um sistema de simples utilização para a geração de compiladores para linguagens em geral. Suas principais características são: a geração automática das classes da AST pelo *JastAdd* e a opção da escolha da 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 uso da programação orientada por aspectos para a implementação das análises semânticas e a geração de código é um outro ponto positivo a favor desse sistema. Contudo, uma das desvantagens do *JastAdd* está relacionada com a não utilização de uma linguagem orientada por aspectos para Java, por exemplo, *AspectJ*. Se o *JastAdd* utilizasse uma linguagem como essa, maiores benefícios seriam alcançados, como:

- seria possível utilizar as facilidades de criação das classes da AST e de sua construção oferecidas pela ferramenta *JJTree* ou por outras ferramentas, pois a inserção estática do código criado pelo usuário nas classes geradas automaticamente seria feita pelo compilador do *AspectJ*, não havendo a necessidade portanto do desenvolvedor alterar os códigos gerados pela ferramenta;

- a inserção estática teria maior flexibilidade. Em *AspectJ* e em outras linguagens orientadas por aspectos, é possível fazer a inserção estática de um mesmo código em mais de uma classe;

- seria possível utilizar ferramentas que auxiliam a programação orientada a aspectos, por exemplo, o *Eclipse* [27].

A única desvantagem de se utilizar o *AspectJ* é a impossibilidade de se ter módulos declarativos. Os módulos declarativos, apesar de possibilitarem a criação de uma quantidade menor de linhas de código, são mais difíceis de serem utilizados do que os módulos imperativos, pois a linguagem que

implementa o RAG, além de ser de menor legibilidade, obriga os usuários a conhecê-la, como também a sua tradução para Java.

JastAdd influenciou o arcabouço deste trabalho em relação a utilização da inserção estática da programação orientada por aspectos para a implementação da análise semântica e geração de código.

3.3 Arcabouço SableCC

O *SableCC* [29], como o *JastAdd*, é um arcabouço para a geração de compiladores implementados em Java para linguagens em geral.

Esse arcabouço gera automaticamente algumas partes e classes do compilador. As partes e as classes são: analisador léxico e sintático, as classes dos nodos da AST e mais algumas classes relacionadas ao padrão Visitor e ao caminhamento da AST.

A análise semântica e a geração de código são implementadas pelo usuário usando uma versão estendida do padrão Visitor que permite maior extensibilidade, uma vez que resolve o problema de adição de novas classes de nodos da AST.

3.3.1 Arquitetura do SableCC

A Figura 3.6 mostra a arquitetura do *SableCC*. Os componentes que estão dentro do retângulo pontilhado são os gerados pelo arcabouço e os demais - *input file*, *Main* e *Visitors* - são criados pelo usuário.

O *SableCC*, diferentemente dos arcabouços *JastAdd* e do *Polyglot*, possui o seu próprio gerador de analisadores léxicos e sintáticos. Portanto não precisa de outros geradores como o *JavaCC*, *JFlex*, *CUP*, etc.

O arquivo de entrada (*input file*) é um arquivo texto com as definições léxicas e as produções gramaticais, a partir dos quais o *SableCC* gera os analisadores léxico e sintático respectivamente. Essas definições, diferente das definições de outras ferramentas, não possuem ação semântica associada a um terminal ou a uma produção da gramática. 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 *input file*. As classes da AST são também geradas automaticamente pelo arcabouço, que usa apenas as definições dos terminais e das produções gramaticais para criá-las.

O *SableCC* ainda gera, a partir das definições do *input file*, a interface *Analysis* e as classes *AnalysisAdapter*, *DephtFirstAdapter* e *ReversedDepthFirstAdapter*, relacionadas ao padrão Visitor e ao caminhamento da AST. O

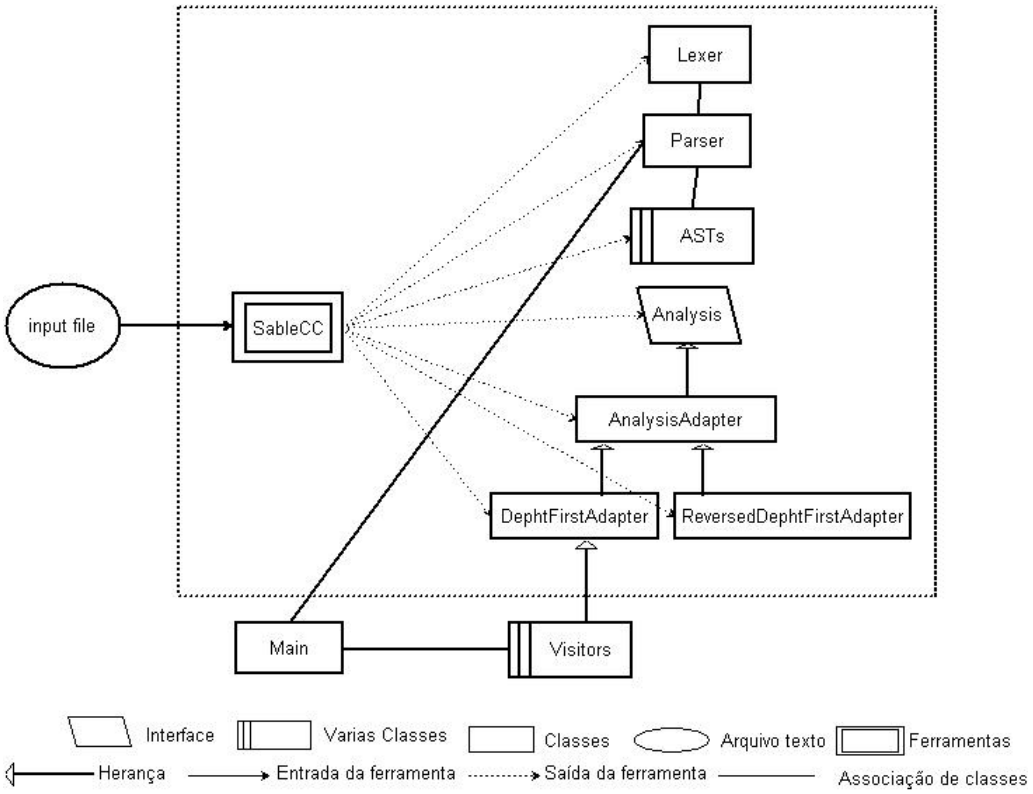

Figura 3.6: Arquitetura do SableCC

usuário é responsável pela implementação das classes de análise semântica e da geração de código, usando herança das classes geradas.

3.3.2 Extensão do Padrão Visitor

O padrão Visitor, como mostrado na Seção 2.2.1, não é adequado para ser aplicado a uma hierarquia de classes que seja estendível. Isto acontece porque, se for adicionada uma nova classe na hierarquia, todas as classes pertencentes ao Visitor precisam adicionar um novo método para que seja possível usá-lo na nova classe, violando o princípio "open-close"de B. Meyer [52, 51]. Mas o *SableCC* resolve este problema. Ele estende o padrão Visitor para que seja possível estender a hierarquia de classe sem a necessidade de alterar as classes já existentes do Visitor.

Para que isso seja possível, são definidas duas interfaces: *Switch* e *Switchable*. A interface *Switch* é a mais genérica de todas as interfaces do Visitor, e a interface *Switchable* é a mais genérica de todas as classes da hierarquia em que o Visitor se aplica. Essas interfaces são mostradas na Figura 3.7 e

um pequeno exemplo de sua utilização é apresentado na Figura 3.8.

```
interface Switch {}
interface Switchable { void accept(Switch sw); }
```

Figura 3.7: Interface Switch e Switchable

```
interface Visitor extends Switch {
    void visitConcreteElementA(ConcretElementA obj);
    void visitConcreteElementB(ConcretElementB obj);
}

abstract class Element implements Switchable { ... }

class ConcreteElementA extends Element {
    ...
    void accept(Switch sw) {
        ((Visitor) sw).visitConcreteElementA(this);
    }
}
```

Figura 3.8: Exemplo da extensão do Visitor

Para adicionar uma nova classe na hierarquia, deve-se criar uma nova interface que estende *Switch*, incluindo o método para essa nova classe. Por exemplo, a Figura 3.9 mostra a adição da classe *ConcreteElementC*. Observe que não há necessidade de alterar as classes da Figura 3.8.

3.3.3 Análises Semânticas e Geração de Código

SableCC define a interface *Analysis* que estende a interface Switch. Essa interface contém todos os métodos necessários para a aplicação do Visitor em todas as classes da AST. A classe *Node*, superclasse de todas as classes da AST, implementa a interface *Switchable*.

A classe *AnalysisAdapter* implementa a interface *Analysis* e providencia uma implementação *default* para todos os métodos.

As classes *DephtFirstAdapter* e *ReversedDephtFirstAdapter* são subclasses de *AnalysisAdapter*. Essas classes adicionam o caminhamento na AST. A classe *DephtFirstAdapter* visita os nodos em profundidade e a classe *ReversedDephtFirstAdapter* visita os nodos em ordem reversa.

Para a implementação de análise semântica e da geração de código, o usuário deve implementar novas subclasses da classe *DephtFirstAdapter* ou

```
interface ExtendedVisitor extends Switch {
    void visitConcreteElementC(ConcretElementC obj);
}

class ConcreteElementC extends Element {
    ...
    void accept(Switch sw) {
        ((ExtendedVisitor) sw).
            visitConcreteElementC(this);
    }
}
```

Figura 3.9: Exemplo da extensão do Visitor nova classe

da *ReversedDephtFirstAdapter*. Ele então deve sobreescrever os métodos que possuem as implementações *default*, implementadas na classe *AnalysisAdpter*, implementando-os de modo adequado para a análise desejada.

3.3.4 Conclusão

SableCC é um arcabouço de fácil utilização para geração de novos compiladores para linguagens em geral. Um dos principais benefícios é a geração automática de várias partes do compilador a partir de uma única definição.

O fato da criação das classes da AST e a própria construção da AST serem definidos usando apenas as definições dos terminais e das produções gramaticais, tem a vantagem de evitar erros provocados pelo usuário, caso o mesmo tenha que especificar as classes da AST e o modo de sua construção. Contudo, esta facilidade proporciona uma menor flexibilidade para o usuário, que fica obrigado a usar as classes definidas pelo SableCC.

Além disso, a utilização do padrão de projeto Visitor é um dos problemas para o arcabouço. A extensibilidade proposta pelo *SableCC* não resolve os problemas mostrados na Seção 2.2.1.

Comparando-o aos outros dois arcabouços definidos nesse capítulo, o *SableCC* apresenta uma maior facilidade de uso e é o que auxilia mais o usuário no processo de desenvolvimento de um compilador.

SableCC influenciou este trabalho em relação à facilidade de uso do arcabouço e no modo de criação automática das classes de nodos da AST.

3.4 Conclusão

Este capítulo apresentou os arcabouços *Polyglot*, *JastAdd* e *SableCC*. Estes arcabouços possuem características diferentes em relação ao:

- uso de geradores de analisadores léxicos e sintáticos;
- implementação dos analisadores semânticos e geração de código;
- construção das classes da AST;
- geração da AST;
- facilidade de uso;
- objetivos.

Esses arcabouços foram usados como referência para o desenvolvimento do Arcabouço proposto por essa dissertação. As características de cada arcabouço listadas acima foram analisadas para que o sistema proposto seja flexível para projetos de linguagens e seja de fácil utilização para o usuário final.

JastAdd teve a influência na utilização da inserção estática da programação orientada por aspectos para a implementação da análise semântica e geração de código. *SableCC* influenciou em relação a facilidade de uso do arcabouço e inspirou o modo de criação automática das classes de nodos da AST, porém o uso do padrão de projeto Visitor foi descartado. E o *Polyglot* não teve influência neste trabalho, pois os objetivos são diferentes.

Capítulo 4

Arcabouço de Compilação para Linguagens de Especificação ASM

Este capítulo discute sobre o Arcabouço proposto, descreve sua arquitetura, sua linguagem de definição para o gerador de front-end. Apresenta também uma proposta para a implementação de análise semântica e geração de código usando programação orientada por aspectos. O capítulo apresenta fragmentos com alguns exemplos de código que demonstram a utilização do Arcabouço. Um exemplo completo encontra-se no Apêndice B.

4.1 *ACOA*: Arcabouço de Compilação Orientado por Aspectos

O *ACOA* é um arcabouço para implementação de compiladores, cujos objetivos são facilitar a geração de compiladores para novas linguagens, e permitir fácil manutenção dos compiladores quando ocorrem mudanças na definição da linguagem. A decisão para a construção deste arcabouço se deve a dois fatos: (1) obter os benefícios das ferramentas conhecidas como *Compiler Compilers*, que automatizam etapas do desenvolvimento de um compilador por meio de definições escritas em declarações de alto nível de abstração; (2) obter maior extensibilidade, flexibilidade e modularização nas etapas implementadas pelo usuário, possibilitando maior evolução dos compiladores desenvolvidos.

O *ACOA* gera compiladores escritos em C++, utilizando a programação orientada por aspectos, via a linguagem AspectC++, para a implementação da análise semântica e geração de código.

O *ACOA* possui as seguintes características:

- os analisadores léxico e sintático são gerados automaticamente;
- as classes dos nodos da AST são geradas automaticamente;
- o analisador sintático gerado cria automaticamente a AST;
- a análise semântica pode ser separada em vários passos de compilação;
- a implementação dos passos de compilação e a geração de código são feitos em métodos que são inseridos estaticamente nas classes da AST por meio da inserção estática da programação orientada por aspectos;
- o uso da programação orientada por aspectos 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 em duas etapas. Na primeira etapa, o usuário constrói um arquivo de especificação que possui em alto nível de abstração as especificações léxicas, a gramática da linguagem, as ações semânticas para a geração de nodos da AST e as declarações e definição da ordem de execução dos passos de compilação. Esse arquivo é similar aos arquivos usados para os *compiler compilers*. O arquivo de especificação é escrito utilizando a linguagem de especificação para o *front-end* definido na Seção 4.4.

A partir desse arquivo de especificação, o *ACOA* gera automaticamente parte da implementação do compilador por meio da utilização do gerador de *front-end FrEG*. O *FrEG*(*Front End Generator*)é uma ferramenta proposta neste trabalho. As partes geradas são: o analisador léxico, que é obtido a partir das especificações léxicas; o analisador sintático, que é obtido a partir da gramática e das ações semânticas para a geração de nodos da AST; classes que representam os nodos da AST e que são gerados a partir de parte da definição léxica, de parte da gramática e das ações semânticas para a geração de nodos da AST; o aspecto *Pass* e a classe *Walk* que são gerados a partir da declaração dos passos de compilação.

O analisador léxico lê uma seqüência de caracteres do arquivo fonte e a organiza como uma seqüência de caracteres que possui algum significado, ou seja, a organiza em *tokens*. O analisador léxico, ao formar um *token*, retorna a constante inteira que representa o *token* formado para o analisador sintático. O analisador sintático recebe essas constantes, construindo produções gramaticais. Para cada produção gramatical construída, o analisador sintático cria, como ação semântica, o nodo da AST que representa

essa produção. A criação do nodo se deve à instanciação de uma classe de um nodo da AST. O final da análise sintática tem como resultado a produção da AST que representa o código especificado no arquivo fonte.

As classes de nodos da AST geradas são de três tipos: as que representam os terminais, as que representam os não-terminais e as que representam as produções gramaticais. A definição e a importância de cada tipo de classe são mostradas na Seção 4.4. Toda classe da AST gerada é subclasse da classe *Node*. A classe *Node* é uma classe pré-definida, que é gerada pelo *ACOA* independente das especificações feitas pelo usuário. A classe *Node* não possui nenhum membro ou método, mas tem sua importância na implementação dos passos de compilação como mostrado a seguir.

O aspecto *Pass* é gerado a partir das declarações dos passos de compilação no arquivo de especificação. O aspecto *Pass* possui a definição de um método puramente virtual para cada passo de compilação declarado. Esses métodos são inseridos estaticamente na classe *Node*. Por exemplo, se o usuário declarou que o compilador possui três passos de compilação chamados de *A*, *B* e *C*, o aspecto *Pass* insere na classe *Node* os três métodos mostrados abaixo:

```
virtual void A() = 0;
virtual void B() = 0;
virtual void C() = 0;
```

Conseqüentemente, todas as classes da AST herdam esses métodos, obrigando ao usuário na próxima etapa da construção do compilador a redefinição desses métodos para cada classe de nodo da AST.

A classe *Walk* inicializa o caminhamento na AST para cada passo de compilação, na ordem definida no arquivo de especificação.

A segunda etapa para a construção de um compilador é a implementação dos passos de compilação. Para implementar os passos de compilação, o usuário deve utilizar a inserção estática pelas definições de aspectos. Considerando o exemplo anterior, o usuário deve portanto prover a implementação especifica dos métodos *A()*, *B()* e *C()* para cada classe de nodo da AST.

4.2 Arquitetura do Arcabouço

Figura 4.1 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, e os demais, *Input File* e *Aspects*, devem ser implementados pelo usuário.

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

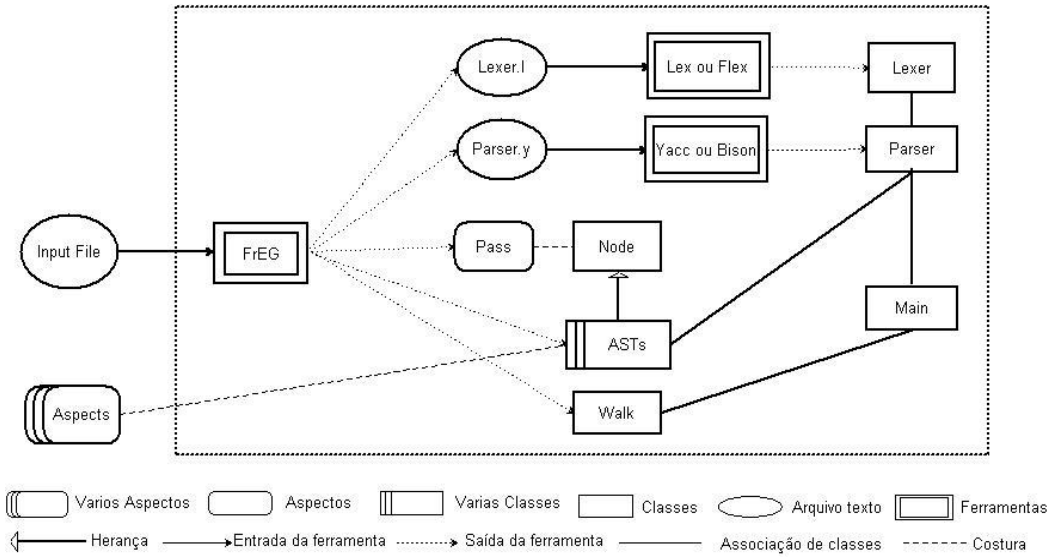

Figura 4.1: Arquitetura do Arcabouço

- ***Input File***: arquivo definido pelo usuário. Nesse arquivo, são definidos a estrutura léxica e a gramática de L, as ações semânticas de geração de nodos da AST e sua construção, e os passos de compilação do compilador de L e sua ordem de execução.

- ***FrEG***(*Front End Generator*): ferramenta proposta neste trabalho, que a partir das definições feitas no arquivo de entrada gera automaticamente as seguintes saídas:

  - ***Lexer.l***: arquivo com a definição do analisador léxico de L no formato adequado para ser usado pelo *Lex* [49] ou *Flex* [59].
  - ***Parser.y***: arquivo com a definição da gramática de L e ações semânticas no formato adequado para ser usado no *Yacc* [43] ou *Bison* [25]. 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. Cada método da classe cria um nodo diferente. Essa classe é usada no analisador sintático (Parser) para construir a AST.
  - ***makefile***: arquivo gerado opcionalmente pelo *FrEG* que não foi mostrado na Figura 4.1. O *makefile* contém a costura 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*.

- ***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. Para o compilador de *L*. O usuário pode também opcionalmente inserir outros métodos e membros nas classes abstratas e concretas da AST.

4.3 Descrição Sintática

A gramática do arquivo de entrada do *FrEG* utiliza a notação BNF estendida [73] ou XBNF. As construções usadas são semelhantes à BNF tradicional, com as seguintes extensões:

|	Opções
$\{X\}$	Zero ou mais ocorrências de X
$[X]$	Zero ou uma ocorrência de X

Tabela 4.1: Extensões da BNF

Os símbolos terminais da gramática são escritos entre aspas.

A sintaxe da entrada do *FrEG* é mostrada ao longo da Seção 4.4 e na sua forma completa, no Apêndice A.

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 *Small* [37] cuja gramática na notação BNF estendida é mostrada na Figura 4.2.

```
Prog     ::= "program" DeclList Com
DeclList ::= Decl ";" DeclList
           |
Decl     ::= "const" Type Identifier "=" Exp
           |  "var" Type Identifier "=" Exp
           |  "proc" Identifier "(" Type Identifier ")" ";"
              DeclList Com
           |  "func" Identifier "(" Type Identifier ")"
              ":" Type ";" DeclList "Begin Exp "End"
Type     ::= "Int"
           |  "Bool"
ComList  ::= Com ";" ComList
           |
Com      ::= Identifier ":=" Exp
           |  "while" Exp "do" Com
           |  "if" Exp "then" Com "else" Com
           |  "Output" "(" Exp ")"
           |  Identifier "(" Exp ")"
           |  "Begin" ComList "End"
Exp      ::= ConstantInt
           |  "true"
           |  "false"
           |  "Read"
           |  Identifier
           |  Exp "+" Exp
           |  Exp "-" Exp
           |  Exp "*" Exp
           |  Exp "/" Exp
           |  Exp "<" Exp
           |  Exp "=" Exp
           |  Exp "and" Exp
           |  Identifier "(" Exp ")"
           |  "if" Exp "then" Exp "else" Exp "End"
           |  "(" Exp ")"
```

Figura 4.2: Gramática de Small

4.4 A Linguagem de Especificação do *Front-End*

Esta seção descreve o processo de preparação de uma especificação para o gerador de front-end (*FrEG*). O arquivo usado como entrada para o *FrEG* é dividido em três partes:

- especificações do analisador léxico (*lexer*) e dos nodos da AST para os *Tokens*;
- especificações do análisador sintático (*parser*) e dos nodos da AST para as regras gramaticais;
- declarações dos passos;

cuja sintaxe é:

```
Start_Grammar    ::= Lexer_Def Parser_Def Passes_Def
Lexer_Def        ::= "%%Lexer" Lexer_Rules
Parser_Def       ::= "%%Parser" Parser_Rules
Passes_Def       ::= "%%Passes" {Pass}
```

4.4.1 Elementos Básicos

Identificadores:

```
Identifier   ::= letter { "_" | letter | digit}
Digit        ::= "0" | "1" | "2" | "3" | "4"
              |  "5" | "6" | "7" | "8" | "9"
Letter       ::= "a" | "b" | "c" | "d" | "e"
              |  "f" | "g" | "h" | "i" | "j"
              |  "k" | "l" | "m" | "n" | "o"
              |  "p" | "q" | "r" | "s" | "t"
              |  "u" | "v" | "w" | "x" | "y"
              |  "z" | "A" | "B" | "C" | "D"
              |  "E" | "F" | "G" | "H" | "I"
              |  "J" | "K" | "L" | "M" | "N"
              |  "O" | "P" | "Q" | "R" | "S"
              |  "T" | "U" | "V" | "W" | "X"
              |  "Y" | "Z"
```

Símbolos:

```
Symbol  ::= """ | "\" | "[" | "]" | "^" | "-" | "?"
         |  "." | "*" | "+" | "|" | "(" | ")" | "$"
         |  "/" | "{" | "}" | "%" | "<" | ">" | "!"
         |  "@" | "#" | "&" | "_" | "=" | "," | ";"
         |  ":" | "~" | "\"
```

Números:

```
Number ::= Digit {Digit}
```

String:

```
String  ::= """ {Letter | Digit | Symbol} """
```

Comentários: Os comentários podem ser usados em qualquer uma das partes do arquivo de entrada para o *FrEG*. Existem dois tipos de comentários:

- que se inicia com /* e termina com */;
- que se inicia com // e termina no final da linha.

4.4.2 Definições para o Analisador Léxico

A definição para o analisador léxico começa com a palavra-chave *%%Lexer*, seguida de regras que podem ser sinônimos, condições e padrões, como mostra a seguinte sintaxe:

```
Lexer_Def           ::= "%%Lexer" Lexer_Rules
Lexer_Rules         ::= {Synonymous} [Conditions] [Patterns]
```

Sinônimos: dão nomes às expressões regulares, conforme ilustrado na Figura 4.3.

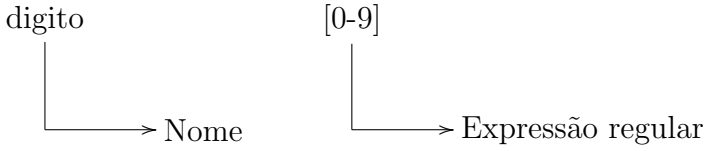

Figura 4.3: Sinônimos

A partir dessa declaração, pode-se usar o nome *digito* entre chaves no lugar de [0-9]. A sintaxe dos sinônimos é a seguinte:

```
Synonymous          ::= Name Regular_Expression {Regular_Expression}
Name                ::= Identifier
```

As expressões regulares possuem a seguinte sintaxe:

```
Regular_Expression ::= String
                   |  "[" {Letter | Digit | Symbol} "]"
                   |  "."
                   |  Regular_Expression "*"
                   |  Regular_Expression "+"
                   |  Regular_Expression "?"
                   |  Regular_Expression
                      "{"number ( ","] | "," number] )"}"
                   |  "{" Name "}"
                   |  "\0"
                   |  "\123"
                   |  "\x2a"
                   |  "(" Regular_Expression ")"
                   |  Regular_Expression "|" Regular_Expression
                   |  Regular_Expression "/" Regular_Expression
                   |  "^" Regular_Expression
                   |  Regular_Expression "$"
                   |  "<" Name ">" Regular_Expression
                   |  "<" "*" ">" Regular_Expression
                   |  "<<" "EOF" ">>"
```

As expressões usadas no *FrEG* obedecem às mesmas regras e convenções das expressões usadas no *Lex* ou *Flex*, portanto mais detalhes sobre as expressões regulares podem ser encontrados em [49, 46].

Padrões: os padrões são usados para definir os *tokens* e suas ações semânticas. A Figura 4.4 mostra o exemplo de um padrão.

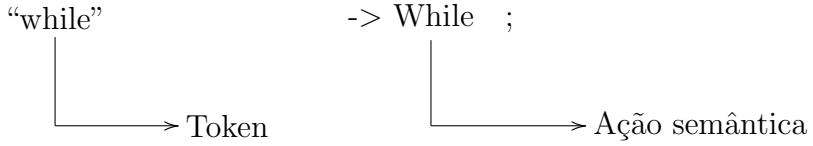

Figura 4.4: Padrões

Os *tokens* são definidos via expressões regulares. As ações semânticas aparecem à direita da definição do *token*. Existem cinco tipos de ações semânticas:

- Ignora o *token*, por exemplo: `[ \n\t] ;`

  O exemplo define que os espaços sejam ignorados.

- Define o *token* com um erro léxico. Por exemplo: `. -> ERROR;`

  O exemplo define que qualquer caractere com exceção do $\backslash n$, definição dada pelo “.”, seja um erro léxico.

- Define o *token* como um dos terminais da gramática, por exemplo:

```
"while"                      -> While;
"*"                          -> mult;
[A-Za-z](_|[A-Za-z]|[0-9])* -> Identificador;
```

O *FrEG* determina que o nome a direita da "->" é o nome do terminal que aparecerá nas regras gramaticais.

No exemplo a palavra-chave "while"é representada na gramática pelo terminal *While*, o símbolo "*"é representado pelo terminal *mult* e os *tokens* que começam com uma letra seguido por nenhum ou vários underscores, letras ou dígitos são representado pelo terminal *Identificador*.

- Ativa ou desativa uma condição. Por exemplo:

```
"/*"                -> BEGIN(comentarios);
<comentarios>"*/"   -> END();
```

No exemplo, é definido que, ao ser encontrado o */**, é ativada a condição *comentarios*, que permanece ativa até que seja encontrada o **/*. A condição é ativada pelo comando *BEGIN* e desativada pelo comando *END*. Uma condição também pode ser desativada quando outra condição é ativada.

- Adiciona o contador de linhas do código fonte. Por exemplo:

```
[\n]               -> endline;
```

O exemplo define que, ao ser encontrado o $\backslash n$, o contador de linhas do código fonte é adicionado. O número de linhas do código fonte é usado nas mensagens de erros léxicos e sintáticos e também pode ser usado pelo usuário nas mensagens de erros semânticos.

As três últimas ações mostradas podem estar presentes na mesma definição de um *token*. Elas devem ser separadas por vírgulas e se o nome do terminal estiver presente, deve ser a última ação declarada.

A sintaxe dos Padrões é a seguinte:

```
Patterns            ::= "%Pattern" Pattern {Pattern}
Pattern             ::= Token_Def [Action] ";"
Token_Def           ::= Regular_Expression {Regular_Expression}
Action              ::= "->" Begin_End_Conditons ["," EndLine]
                             ["," Name_Token_AST]
                      |  "->" EndLine ["," Begin_End_Conditons]
                             ["," Name_Token_AST]
```

```
                      |  "->" Name_Token_AST | "->" Error
Begin_End_Conditons ::= "BEGIN" "(" Name ")"
                      | "END" "(" ")"
EndLine             ::= "endline"
Error               ::= "ERROR"
Name_Token_AST      ::= Token_Type
Token_Type          ::= Terminal
```

Condições: é o mecanismo condicional para a formação de *tokens*. Qualquer padrão que comece com $<c>$ somente será formado se a condição de nome c estiver ativa. Por exemplo:

```
<comentarios>[^\n]*                ;
<comentarios>"\n"          -> endline;
```

Se a condição *comentarios* estiver ativa, o primeiro padrão ignora todos os caracteres com a exceção do $\backslash n$ e o segundo padrão ao encontrar ao $\backslash n$ adiciona o contador de linhas do código fonte.

A declaração das condições começa com a palavra-chave *%inclusive* ou *%exclusive* seguido por uma lista de nomes, por exemplo:

```
%inclusive comentarios string
%exclusive caracter
```

Se a condição é do tipo *%inclusive*, então as regras sem condição e as regras com uma determinada condição ativa podem ser usadas. Se a condição é do tipo *%exclusive*, somente as regras com uma determinada condição ativa podem ser usadas. A sintaxe das condições é mostrada a seguir:

```
Conditions          ::= "%Conditions" Conditions_list
Conditions_list     ::= Conditions_mode {Conditions_mode}
Conditions_mode     ::= ("%inclusive" | "%exclusive") Name {Name}
```

A Figura 4.5 mostra parte da definição do analisador léxico para a linguagem *Small*. A definição completa é descrita no Apêndice B desta dissertação.

Para todos os terminais definidos, o *FrEG* gera a declaração de uma classe de um nodo da AST correspondente ao terminal. A classe é gerada com o mesmo nome do terminal, possuindo como membro apenas uma string que possui o valor do *token*. Para recuperar esse valor deve-se utilizar o método da classe chamado de *getToken_value()*. Todas as classes deste tipo são subclasses da classe *Token*. A classe *Token* é gerada pelo *ACOA* independente da especificação feita pelo usuário, sendo usada apenas para agrupar em uma hierarquia diferente as classes que representam um terminal.

Vale ressaltar que essas classes em nenhum momento são instanciadas pelo analisador léxico. Sua instanciação acontece no analisador sintático caso o

```
%%Lexer

letra  [A-Za-z]
digito [0-9]
numero {digito}+
identificador {letra} (_ | {letras}{digito})*

%Conditions
%inclusive comentarios

%Pattern
"/*"                 -> BEGIN(comentarios);
<comentarios>"*/"    -> END();
<comentarios>[^*\n]*         ;
"program"            -> program;
"var"                -> Var;
"proc"               -> Proc;
"func"               -> Func;
":="                 -> OAtr;
"*"                  -> mult;
"/"                  -> divi;
{numero}             -> ConstanteInt;
```

Figura 4.5: Parte da definição léxica de Small

usuário tenha especificado que um terminal faça parte de um nodo da AST que representa uma produção gramatical, como será visto na Seção 4.4.3.

Para ilustrar a geração das classes, considere por exemplo os padrões da Figura 4.6, cuja classes são apresentadas na Figura 4.7.

```
"if"             -> If;
"then"           -> Then;
"else"           -> Else;
{numero}         -> ConstanteInt;
{id}             -> Identificador;
```

Figura 4.6: Padrões

4.4.3 Definições para o Analisador Sintático

A definição para o analisador sintático (*parser*) começa com a palavra-chave *%%Parser* seguida por declarações e regras gramaticais. A sintaxe para o analisador sintático é a seguinte:

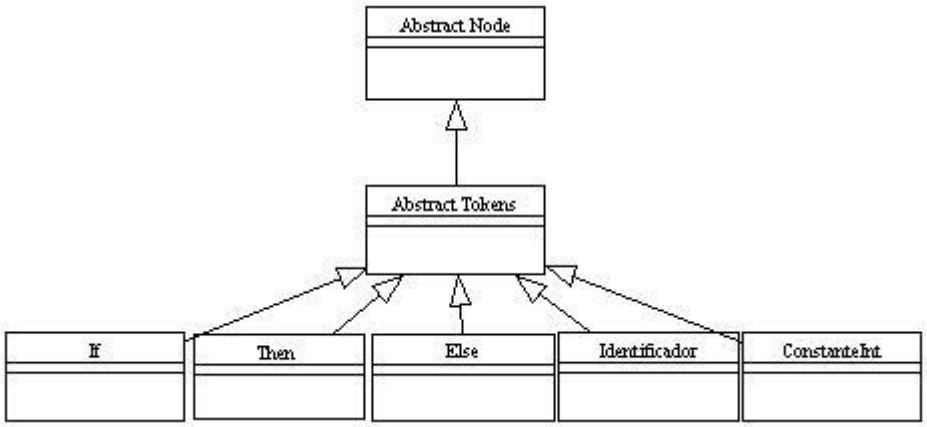

Figura 4.7: Classes da AST

```
Parser_Def      ::= "%%Parser" Parser_Rules
Parser_Rules    ::= Declaration [Grammar]
Grammar         ::= "%Grammar" {Rule}
```

Declarações: o usuário pode fazer 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 definidas pelas palavras-chave *%left*, *%right*, *%nonassoc* seguida por uma lista de terminais ou nomes. Sua sintaxe é:

```
Declaration    ::= {Precedence} [Start]
Precedence     ::= ("%left" | "%right" | "%nonassoc")
                   (Terminal | Name) {Terminal | Name}
Terminal       ::= Identifier
Name           ::= Identifier
```

Todos os símbolos terminais ou nomes contidos na mesma lista possuem o mesmo nível de precedência. As listas mostram os operadores em ordem decrescente de precedência. Por exemplo:

```
%left add sub
%left mult divi
```

descrevem a precedência e associatividade para a adição, subtração, multiplicação e divisão. Todos eles são associativos à esquerda, mas os operadores de adição e subtração têm menor precedência que os operadores de multiplicação e divisão.

O símbolo inicial da gramática é definido pela palavra-chave *%start* seguida pelo não-terminal desejado. Se essa declaração for omitida, o *FrEG* considera por *default* como símbolo inicial o não-terminal do lado esquerdo da primeira regra da gramática.

```
Declaration       ::= {Precedence} [Start]
Start             ::= "%start" Nonterminal
Nonterminal       ::= Identifier
```

Regras gramaticais: Uma regra gramatical para o *FrEG* tem a seguinte sintaxe:

```
Rule  ::= Left_Hand "::=" Right_Hand { "|" Right_Hand} ";"
```

onde *Left_Hand* representa um não-terminal da gramática. Caso existam várias regras gramaticais com o mesmo lado esquerdo, pode-se usar o símbolo "|" para evitar reescrever o lado esquerdo. Conseqüentemente, o ponto e vírgula só aparece na última regra. Por exemplo, considere as seguintes regras gramaticais:

```
A ::= B;
A ::= C;
A ::= D;
A ::= E;
```

elas podem ser escritas para o *FrEG* do seguinte modo:

```
A ::= B
   |  C
   |  D
   |  E ;
```

Não é necessário que todas as regras gramaticais com o mesmo lado esquerdo apareçam juntas separadas pelo símbolo "|", apesar de essa forma ser mais legível e mais fácil de ser alterada.

O *FrEG* gera, a partir dos não-terminais do lado esquerdo da gramática, classes abstratas da AST. Essas classes possuem o mesmo nome dos não-terminais, não possuindo qualquer membro ou método. Por exemplo, para a gramática da Figura 4.8, a Figura 4.9 mostra as classes abstratas geradas.

O usuário pode alterar os nomes das classes abstratas geradas sem a necessidade de alterar o nome dos não-terminais. Para isso, o mesmo deve colocar o símbolo ":" depois do não-terminal seguido pelo nome desejado, como se vê na sintaxe a seguir:

```
Program  ::=   ...;
Com      ::=   ...;
Exp      ::=   ...;
Decl     ::=   ...;
```

Figura 4.8: Gramática

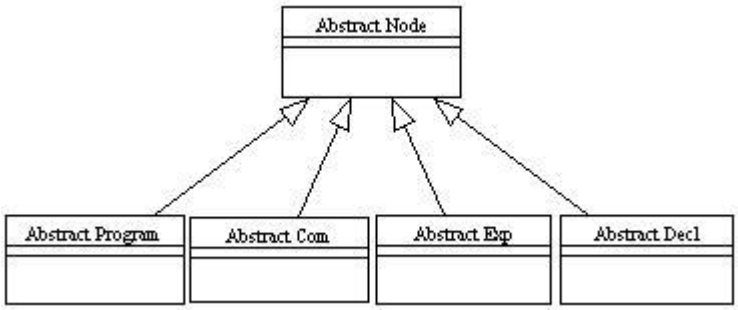

Figura 4.9: Classes da AST

```
Left_Hand           ::= Tagged_Nonterminal
Tagged_Nonterminal  ::= Nonterminal [":"Ast_Id]
Nonterminal         ::= Identifier
Ast_Id              ::= Identifier
```

O objetivo dessa flexibilidade é permitir que o usuário use um nome mais elucidativo nas classes geradas, enquanto que nas regras gramaticais utiliza-se nomes mais simples ou abreviaturas. Para ilustrar, considere a gramática da Figura 4.10, cujas classes abstratas geradas são mostradas na Figura 4.11.

```
Program              ::=   ...;
Com : Comandos       ::=   ...;
Exp : Expressoes     ::=   ...;
Decl : Declaracoes   ::=   ...;
```

Figura 4.10: Gramática

O lado direito de uma regra gramatical, *Right_Hand*, tem a seguinte sintaxe:

```
Right_Hand          ::= Production_Elements [ "->" Ast_Construction ]
Production_Elements ::= {Nonterminal | Terminal}
                      |  {Nonterminal | Terminal}
                         "%prec" (Name | Terminal)
Terminal            ::= Identifier
```

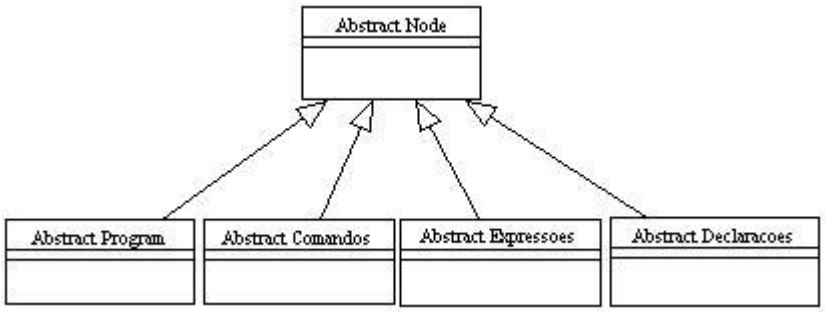

Figura 4.11: Classes da AST

```
Nonterminal          ::= Identifier
Name                 ::= Identifier
```

Production_Elements representa uma seqüência de zero ou mais terminais ou não-terminais, possivelmente decorados com índices. Os índices não alteram o significado dos elementos, servem apenas para diferenciar elementos do mesmo tipo no momento de construir o nodo da AST. Os índices possuem apenas um dígito. Se for necessário mais de um dígito para diferenciar os elementos a produção deve ser dividida em duas ou mais produções.

Ao encontrar um elemento em uma produção, o *FrEG* faz a seguinte busca para verificar se esse elemento é um terminal ou não-terminal:

- verifica se existe um terminal ou não-terminal com o mesmo nome do elemento.
- Se não existir, o *FrEG* verifica se o elemento possui um índice.
- Se o elemento possuir um índice, o *FrEG* verifica se existe um terminal ou não-terminal com o mesmo nome do elemento sem o índice.
- Caso nenhuma das opções acima encontrar o terminal ou o não-terminal correspondente ao elemento, ele é considerado como não declarado e um erro é produzido.

O *Production_Elements* pode terminar com o operador "*%prec*" seguido de um terminal ou de um nome que tenha sido usado nas declarações de precedências. O operador "*%prec*" altera a precedência da produção. Por exemplo, considere o caso do menos unário e binário. O unário tem maior precedência do que o binário porém os *tokens* de ambos são iguais. Para alterar a precedência é feito o seguinte:

```
...
%%Parser
...
%left sub
...
%left menosUnario
...
%Grammar
...
Exp ::= Exp sub Exp
        sub Exp %prec menosUnario
        ...
...
```

A produção com o menos unário tem a precedência do *menosUnario* e não de *sub*, portanto possui maior precedência.

Ast_Construction representa a ação semântica da regra, que por sua vez determina ao analisador sintático a inserção do nodo declarado depois da "->" na AST.

Ao encontrar uma *Ast_Construction*, o *FrEG* gera uma classe concreta de um nodo da AST representando a regra gramatical. A regra *Ast_Construction* tem a seguinte sintaxe:

```
Ast_Construction    ::= Ast_Id "(" [Ast_Elements] ")"
Ast_Id              ::= Identifier
Ast_Elements        ::= Ast_Element {"," Ast_Elements }
Ast_Element         ::= Tagged_Nonterminal | Tagged_Terminal
Tagged_Terminal     ::= Terminal[":"Ast_Id]
Tagged_Nonterminal  ::= Nonterminal [:Ast_Id]
Terminal            ::= Identifier
Nonterminal         ::= Identifier
```

onde, *AST_Id* é o nome da classe, *Ast_Element* é um terminal ou não-terminal 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. Por exemplo, considere a gramática da Figura 4.12, onde *Program*, *Com*, *Exp* são os não-terminais, e o restante dos nomes são terminais. As classes geradas pelo *FrEG* são mostradas na Figura 4.13. Observe que as classes geradas para os terminais não são mostradas nessa figura.

As classes *Program*, *Comandos* e *Expressoes*, mostradas na Figura 4.13, são as classes geradas pelos não-terminais como apresentado anteriormente. As classes definidas pela *Ast_Construction* são subclasses da classe que representa o não-terminal do lado esquerdo da regra gramatical no qual a

```
Program         ::= program Com  -> Inicio(Com);
Com:Comandos    ::= Identificador Oatr Exp
                            -> Atr(Identificador, Exp)
                  |  While Exp Do Com -> WhileComando(Exp, Com)
                  |  If Exp Then Com1 Else Com2
                            -> IfComando(Exp, Com1, Com2);
Exp:Expressoes  ::= ConstanteInt  -> Numero(ConstanteInt)
                  |  True -> ConstanteTrue(True)
                  |  False -> ConstanteFalse(False)
                  |  Identificador -> IdExp(Identificador);
```

Figura 4.12: Gramática

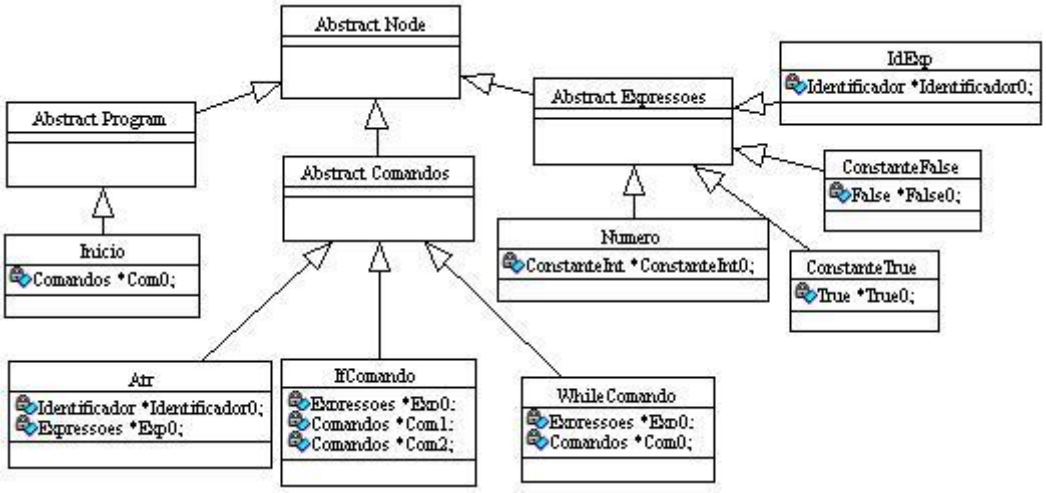

Figura 4.13: Classes da AST

Ast_Construction faz parte. Por exemplo, as classes *Atr*, *WhileComando* e *IfComando* são subclasses da classe *Comandos*.

Note, na Figura 4.13, que os nomes dos membros são os nomes dos terminais e não-terminais seguidos pelo índice. Se os terminais ou os não-terminais não possuem índices na definição da gramática, os nomes dos membros correspondentes aos nomes dos terminais e dos não-terminais são seguidos por quantos zeros forem necessários para que eles se tornem nomes válidos.

Não há obrigatoriedade que todos os índices para os elementos do mesmo tipo dentro de uma mesma regra sejam diferentes, mas se existir algum elemento na *Ast_Elements* que possua mais de um elemento *Production_Elements* com o mesmo índice, isso provocará um erro.

As classes para os não-terminais são geradas porque alguns dos membros das classes concretas precisam ser polimórficos. Essa necessidade aparece na

seguinte regra gramatical da Figura 4.12:

```
Com:Comandos ::= While Exp Do Com -> WhileComando(Exp, Com);
```

O elemento *Com* que aparece do lado direito de "*::=*" e antes da regra semântica pode ser qualquer uma das seguintes regras:

```
Com:Comandos   ::= Identificador Oatr Exp
                        -> Atr(Identificador, Exp)
                 |  While Exp Do Com -> WhileComando(Exp, Com)
                 |  If Exp Then Com1 Else Com2
                        -> IfComando(Exp, Com1, Com2);
```

portanto o membro *Com0* da classe *WhileComando* mostrado na Figura 4.13 necessita referenciar qualquer um dos seguintes nodos: *Atr*, *IfComando*, *WhileComando*. Isso se torna possível devido ao fato de *Cmd0* ser uma referência para o tipo *Comandos*, que é a superclasse das classes desses nodos.

O usuário também pode alterar os nomes dos elementos da *Ast_Elements*, para isso basta usar o símbolo ":" depois de um elemento e em seguida usar o nome desejado.

Por exemplo, considere as alterações da gramática 4.12 mostrada na Figura 4.14. As classes geradas pelo *FrEG* são mostradas na Figura 4.15. Desse modo, os nomes dos membros ficam mais legíveis e mais simples de serem usados posteriormente.

```
Program      ::= program Com  -> Inicio(Com:Comando_inicial);

Com:Comandos ::= Identificador Oatr Exp
          -> Atr(Identificador:id, Exp:Lado_dir)
        |  While Exp Do Com
          -> WhileComando(Exp:Condicao, Com:Com_While)
        |  If Exp Then Com1 Else Com2
          -> IfComando(Exp:Condicao, Com1:Com_Then, Com2:Com_Else);

Exp:Expressoes ::= ConstanteInt  -> Numero(ConstanteInt)
                |  True -> ConstanteTrue(True)
                |  False -> ConstanteFalse(False)
                |  Identificador -> IdExp(Identificador:id);
```

Figura 4.14: Gramática

O *FrEG* gera também métodos *set* e *get* para todas as classes da AST que possuem algum membro, e os métodos *_Signature* e *_getPosCodeSource* para todas as classes. O método *_Signature* retorna o nome da classe e o

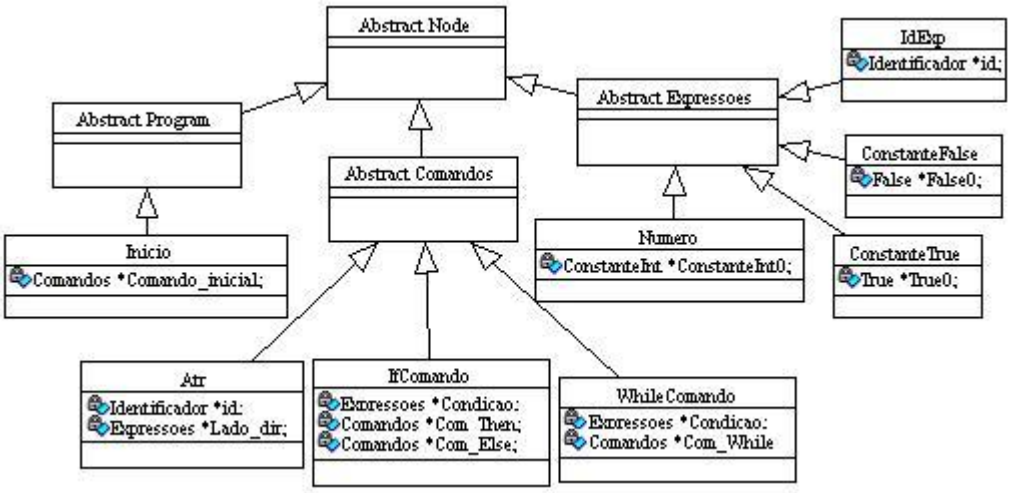

Figura 4.15: Classes da AST

método *_getPosCodeSource* retorna a posição da produção no código fonte. Para simplificar as figuras anteriores, esses métodos não foram mostrados.

O Apêndice B apresenta a definição completa do analisador sintático para a linguagem *Small*, definida na Figura 4.2.

4.4.4 Declaração dos Passos

A definição e a ordem dos passos são feitas começando pela palavra-chave *%%Passes*, seguida por uma lista de nomes dos passos. A ordem em que aparecem os nomes determina a ordem dos passos de compilação. Por meio dessa definição, o *FrEG* gera o aspecto *Pass* e a classe *Walk*. Os nomes dos passos correspondem aos nomes dos métodos inseridos nas classes dos nodos da AST pelo aspecto *Pass*. A sintaxe dos passos é a seguinte:

```
Passes_Def      ::= "%% Passes" {Pass}
Pass            ::= Name_Pass
Name_Pass       ::= Identifier
```

Para ilustrar essa construção, considere por exemplo os passos *verificaTipo* e *geraCodigo*, onde o passo *verificaTipo* é executado antes de *geraCodigo*. As declarações desses passos para o *FrEG* são feitas da seguinte forma:

```
verificaTipo geraCodigo
```

O aspecto *Pass*, mostrado na Figura 4.16, introduz estaticamente na classe *Node* métodos abstratos que correspondem aos passos do compilador.

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 detectar um erro no momento de sua compilação, devido ao fato de existir uma tentativa de instanciar uma classe abstrata. A classe *Walk* inicia o caminhamento na AST para cada passo do compilador. Ela possui um único método chamado *start* que recebe o nodo raiz da AST e ativa os métodos que implementam os passos para este nodo.

Para entender a Figura 4.16, 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 e os membros e métodos que aparecem dentro de um retângulo em uma classe indicam que os mesmos foram inseridos estaticamente por um aspecto.

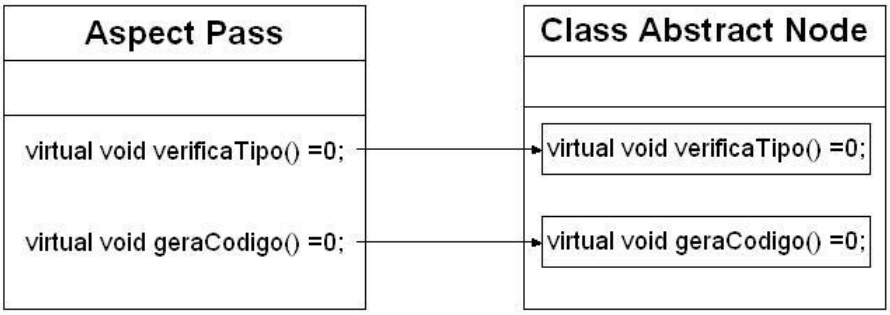

Figura 4.16: Aspecto Pass

```
#include "ASTs.h"
#include "Walk.h"

Walk::Walk() { }
Walk::~Walk() { }
void Walk::start(Node *raiz) {
    raiz->verificaTipo();
    raiz->geraCodigo();
}
```

Figura 4.17: Walk.cpp

A classe *Walk*, mostrada na figura 4.17, inicia o caminhamento na AST para cada passo do compilador. Ela possui um único método chamado *start*

que recebe o nodo raiz da AST e ativa os métodos que implementam os passos para este nodo.

4.5 Implementação dos Passos de Compilação

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

Para a implementação dos passos de compilação, foram estudadas duas soluções usando aspectos: a primeira delas usa pontos de junção e a segunda usa inserção estática. Neste trabalho de dissertação, adotou-se pela segunda solução, o uso de inserção estática. Esta seção descreve cada uma delas e apresenta os argumentos que justificam o uso da segunda solução.

4.5.1 Solução Usando Pontos de Junção

Esta solução modifica a saída do *FrEG* do seguinte modo:

- não é mais gerado o aspecto Pass.
- As classes concretas da AST, ao serem geradas, já possuem os métodos que implementam os passos de compilação. Dentro desses métodos, o *FrEG* providencia apenas o caminhamento na AST. As Figuras 4.18 e 4.19 mostram como seria a implementação da classe *IfComando* para o comando *if* da linguagem Small (veja Seção 4.3).
- O *FrEG* gera os aspectos abstratos que possuem os pontos de junção das classes concretas da AST. Cada aspecto determina os pontos de junção de uma classe da AST. Por exemplo, o aspecto gerado pelo *FrEG* para a classe *IfComando* é mostrado na Figura 4.20

Note que o usuário precisa implementar os aspectos concretos que possuem regras de junção para os pontos de junções herdados dos aspectos abstratos. Essas regras de junção possuem os códigos necessários para finalizar a implementação de cada passo de compilação. A Figura 4.21 mostra uma simples implementação para a verificação de tipo e geração de código[1] para a classe *IfComando*.

Algumas vantagens dessa solução são:

[1] o código é gerado na linguagem C

```
#ifndef _IFCOMANDO_H_
#define _IFCOMANDO_H_

#include Asts.h"

class IfComando : public Comandos {
private:
    Expressoes *Condicao;
    Comandos *CmdThen;
    Comandos *CmdElse;

public:
    IfComando(Expressoes *, Comandos *, Comandos *);
    virtual ~IfComando();
    string _Signature();
    Expressoes* getCondicao();
    Comandos* getCom_Then();
    Comandos* getCom_Else();
    void setCondicao(Expressoes *);
    void setCom_Then(Comandos *);
    void setCom_Else(Comandos *);
    void verificaTipo();
    void geraCodigo();
};

#endif //_IFCOMANDO_H_
```

Figura 4.18: IfComando.h

- implementação *default* para todos os passos de compilação já está pronta para as classes dos nodos da AST.
- É possível acessar membros privados das classes da AST.
- É possível adicionar novos membros e métodos nas classes da AST.

As desvantagens dessa solução são:

- os códigos para os passos ficam muito fragmentados, o que faz com que eles sejam difíceis de serem entendidos e conseqüentemente o arcabouço fica mais difícil de ser usado e de ser mantido.
- Em aspectC++, é possível ter herança no caso do aspecto base ser abstrato. Portanto, o usuário, ao construir o aspecto que herda os pontos de junção do aspecto gerado pelo *FrEG*, teria que definir o método *abstrato()* para que seu aspecto não seja abstrato.

```
#include "IfComando.h"

IfComando::IfComando(Expressoes *e, Comandos *c1, Comandos *c2) {
    Condicao = e;
    Com_Then = c1;
    Com_Else = c2;
}

IfComando::~IfComando() {
    if(Condicao != NULL){
        delete Condicao;
    }
    if(Com_Then != NULL){
        delete Com_Then;
    }
    if(Com_Else != NULL){
        delete Com_Else;
    }
}

string IfComando::_Signature() { return "IfComando";}

Expressoes* IfComando::getCondicao() {return Condicao;}

Comandos* IfComando::getCom_Then() {return Com_Then;}

Comandos* IfComando::getCom_Else() {return Com_Else;}

void IfComando::setCondicao(Expressoes *e) {Condicao = e;}

void IfComando::setCom_Then(Comandos *c) {Com_Then = c;}

void IfComando::setCom_Else(Comandos *c) {Com_Else = c;}

void IfComando::verificaTipo() {
    Condicao->verificaTipo();
    Com_Then->verificaTipo();
    Com_Else->verificaTipo();
}
void IfComando::geraCodigo(){
    Condicao->geraCodigo();
    Com_Then->geraCodigo();
    Com_Else->geraCodigo();
}
```

Figura 4.19: IfComando.cpp

```
#ifndef _AIFCOMANDO_H_
#define _AIFCOMANDO_H_

aspect aIfComando {
  virtual void abstrato() = 0;

  pointcut pverificaTipo() =
    execution("% IfComando::verificaTipo(...)");

  pointcut pExpressoes_vericaTipo() =
    call("% Expressoes::verificaTipo(...)")
    && cflow(pverificaTipo());

  pointcut pComandos_vericaTipo() =
    call("% Comandos::verificaTipo(...)")
    && cflow(pverificaTipo());

  pointcut pgeraCodigo() =
    execution("% IfComandos::geraCodigo(...)");

  pointcut pExpressoes_geraCodigo() =
    call("% Expressoes::geraCodigo(...)")
    && cflow(pgeraCodigo());

  pointcut pComandos_geraCodigo() =
    call("% Comandos::geraCodigo(...)")
    && cflow(pgeraCodigo());
};

#endif
```

Figura 4.20: aIfComando.ah

- Se o usuário se esquecer de definir o método *abstrato()*, no aspecto criado não existará a costura das regras de junção do aspecto com o código da classe da AST.
- Se uma classe da AST possuir membros de um mesmo tipo, o usuário tem que diferenciá-los no momento de construir as regras de junção, como foi mostrado nas linhas 8 e 19 da Figura 4.21.
- Os campos privados das classes da AST podem ser acessados chamando *tjp->that()*, como mostrado na linha 9 da Figura 4.21.
- Se o usuário se esquecer de tratar algum nodo da AST que não deve ter a implementação *default*, esse erro só será percebido quando ele usar o

```
1  #ifndef _ASPECTO_IFCOMANDO_H_
2  #define _ASPECTO_IFCOMANDO_H_
3  ...
4  aspect IfComandoAsp : public aIfComando {
5    void abstrato(){}
6
7    advice pComandos_vericaTipo() : before()  {
8       if(tjp->that()->Com_Then == tjp->target()) {
9          if(tjp->that()->Condicao->gettipo() != boolean)
10           erros->push_back(...);
11      }
12   }
13
14   advice pExpressoes_geraCodigo() : before()  {
15      (*saida) <<"  if(";
16   }
17
18   advice pComandos_geraCodigo() : before()  {
19      if(tjp->that()->CmdThen == tjp->target())
20        (*saida) <<"){\n";
21      else {
22        (*saida) <<"  }\n";
23        (*saida) <<"  else{\n";
24      }
25   }
26
27   advice pExpressoes_geraCodigo() : after()  {
28      (*saida) <<"  }\n";
29   }
30 };
31 #endif
```

Figura 4.21: IfComandoAsp.ah

compilador instanciado.

4.5.2 Solução Usando Inserção Estática

Nessa solução, o usuário insere estaticamente métodos que implementam os passos de compilação nas classes concretas da AST. Para ilustrar essa solução a Figura 4.22 mostra o mesmo exemplo apresentado na Figura 4.21. Observe que, nesse caso, a classe *IfComando* gerada pelo *FrEG* não possui os métodos *verificaTipo()* e *geraCodigo()*.

As vantagens para essa solução são:

- o código é mais claro de ser entendido, tornando o arcabouço mais

```
1  #ifndef _IFCOMANDOASP_AH_
2  #define _IFCOMANDOASP_AH_
3  ...
4  aspect IfComandoAsp {
5
6  public:
7      advice "IfComando" : void verificaTipo() {
8        Condicao->verificaTipo();
9        if(Condicao->gettipo() == boolean) {
10         Com_Then->verificaTipo();
11         Com_Else->verificaTipo();
12       }
13       else
14         erros->push_back(...);
15     }
16
17     advice "IfComando" : void geraCodigo() {
18       (*saida) <<"  if(";
19       Condicao->geraCodigo();
20       (*saida) <<"){\n";
21       Com_Then->geraCodigo();
22       (*saida) <<"  }\n";
23       (*saida) <<"  else{\n";
24       Com_Else->geraCodigo();
25       (*saida) <<"  }\n";
26     }
27
28 };
29
30 #endif
```

Figura 4.22: IfComandoAsp.ah (inserção estática)

simples de ser usado e mantido.

- Não usa herança de aspectos.
- Permite que o usuário organize os aspectos da sua maneira, 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.
- É possível acessar membros privados das classes da AST usando apenas o nome do membro.
- É possível adicionar novos membros e métodos nas classes da AST.

- Se o usuário se esquecer de tratar algum nodo da AST, para qualquer um dos passos, será gerado um erro no momento de compilar o código do compilador gerado pelo arcabouço.

As desvantagens para essa solução são:

- Não existe uma implementação *default*, obrigando o usuário tratar todos os nodos da AST que são instanciados.
- O usuário fica responsável pelo caminhamento na AST.

Para explicar o código da Figura 4.22, é mostrada abaixo a especificação do comando *if* de *Small* para o *FrEG*. Observando essa especificação, pode-se perceber que a classe gerada *IfComando* possui três membros: *Condicao*, que é uma referência para a classe *Expressoes*, *Com_Then* e *Com_Else*, que são referências para a classe *Comandos*.

```
Com:Comandos ::= If Exp Then Com1 Else Com2
          -> IfComando(Exp:Condicao, Com1:Com_Then, Com2:Com_Else);
```

As linhas 7 e 17 instruem o costurador de código a inserir os métodos *verificaTipo()* e *geraCodigo()*, respectivamente, na classe *IfComando*.

Para a construção do compilador de *Small*, todas as subclasses de *Expressoes* tiveram a inserção estática de um novo membro do tipo inteiro chamado de *tipo*, como mostrado na Figura 4.23. Além da inserção do novo membro, essas classes também tiveram a inserção de dois novos métodos, o *gettipo* e o *settipo*, como mostrado na Figura 4.24. Essas novas inserções permitem que os nodos do tipo *Expressoes* armazenem o tipo da expressão.

```
1  ...
2  private:
3     advice "Numero" || "ConstanteTrue" || "ConstanteFalse"
4            || "Adicao" || "Subtracao" || "IdExp"
5            || "Multiplicacao" || "MenorExp"
6            || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
7            || "IfExpressao" || "ChamaFunc" || "Divisao"
8            || "AndExp" : int tipo;
9
10 ...
```

Figura 4.23: ExpressoesAsp.ah

Para exemplificar o uso dessas inserções, considere a enumeração mostrada abaixo, que enumera os tipos de *Small*, e a Figura 4.25.

```
1  ...
2  public:
3
4      advice "Expressoes" : virtual int gettipo() = 0;
5
6      advice "Expressoes" : virtual void settipo(int ) = 0;
7
8      advice "Numero" || "ConstanteTrue" || "ConstanteFalse"
9             || "Adicao" || "Subtracao" || "IdExp"
10            || "Multiplicacao" || "MenorExp" || "Negacao"
11            || "IgualExp" || "ParExp" || "ReadValor"
12            || "IfExpressao" || "ChamaFunc" || "Divisao"
13            || "AndExp" : int gettipo() {
14                return tipo;
15     }
16
17     advice "Numero" || "ConstanteTrue" || "ConstanteFalse"
18            || "Adicao"  || "Subtracao" || "IdExp"
19            || "Multiplicacao" || "MenorExp" || "Negacao"
20            || "IgualExp" || "ParExp" || "ReadValor"
21            || "IfExpressao" || "ChamaFunc" || "Divisao"
22            || "AndExp" : void settipo(int t) {
23                tipo = t;
24     }
25     ...
```

Figura 4.24: ExpressoesAsp.ah

```
enum tipos_valor {boolean, integer, nothing};
```

```
1  ...
2      advice "IgualExp" || "MenorExp" : void verificaTipo() {
3          Exp1->verificaTipo();
4          Exp2->verificaTipo();
5          if(Exp1->gettipo() !=  Exp2->gettipo())
6            erros->push_back(...);
7
8          tipo = boolean;
9      }
10
11 ...
```

Figura 4.25: ExpressoesAsp.ah

A Figura 4.25 mostra a implementação de *verificaTipo* para as expressões de *igualdade* e *menor que*. Nas linhas 3 e 4 as expressões têm seus tipos verificados, a linha 5 verifica se as expressões possuem o mesmo tipo e a linha 8 atribui o tipo booleano ao tipo da expressão.

Voltando ao código da Figura 4.22, a linha 8 verifica o tipo de *Condicao* e a linha 9 verifica se *Condicao* é do tipo booleano. Se o tipo de *Condicao*

é do tipo booleano, então são verificados os tipos dos membros *Com_Then* e *Com_Else*, caso contrario é declarado um erro semântico.

Para o método *geraCodigo*, as linhas 18, 20, 22, 23 e 25 da Figura 4.22 possuem a saída do código gerado para o arquivo de saída e as linhas 19, 21 e 24 representam a chamada da geração de código para os membros das classes de *IfComando*.

Para ilustrar a implementação dos passos de compilação, na Figura 4.26 é esboçada a inserção estática do aspecto *Pass* na classe *Node* e a inserção estática dos métodos definidos na Figura 4.22 para a classe concreta *IfComando*.

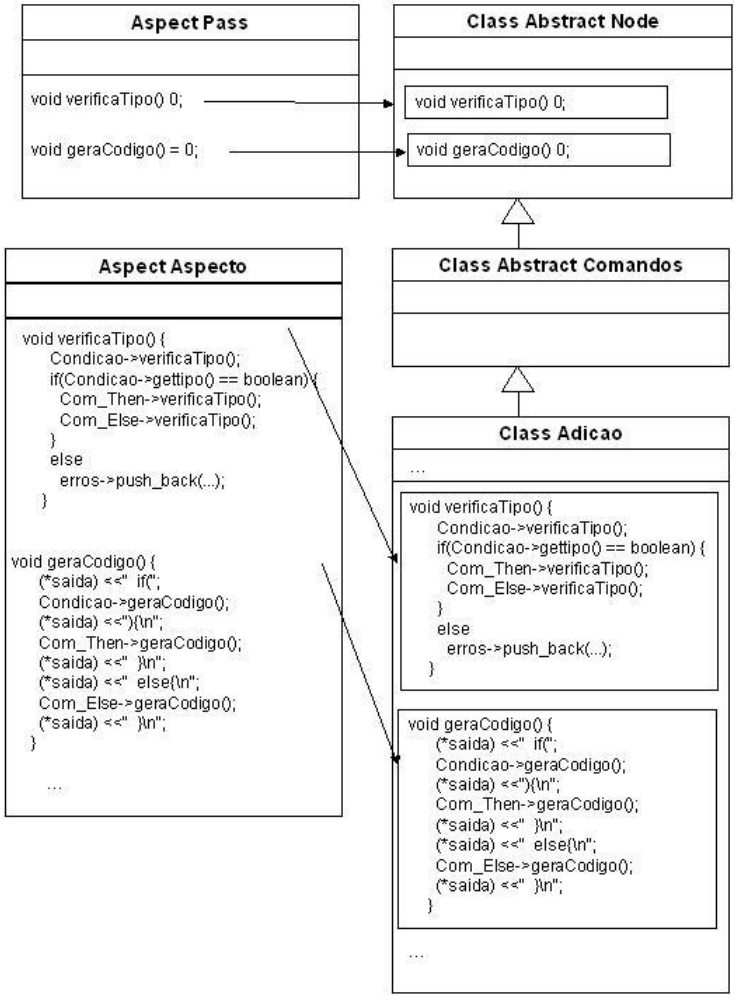

Figura 4.26: Implementação dos passos de compilação

4.5.3 Avaliação das Soluções Propostas

A solução usando inserção estática, além de ter mais vantagens em relação à solução que usa pontos de junção, consegue resolver os problemas associados ao padrão Visitor mostrados na Seção 2.2.1, do seguinte modo [38]:

Problema da confusão: esse tipo de problema não ocorre com inserção 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 diretamente 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.

4.6 Conclusão

Este capítulo 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 projetos de linguagens bastante simples de ser usada. Um dos fatores que se deve a sua simplicidade é a geração automática do analisador léxico, analisador sintático e das classes dos nodos da AST por meio da especificação de alto nível para o *FrEG*, o que permite ao desenvolvedor fazer modificações na sintaxe de sua linguagem.

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

Capítulo 5

Avaliação e Validação dos Resultados

Este capítulo discute a avaliação do *ACOA*, destacando a facilidade de uso e principalmente a flexibilidade proporcionada pelo arcabouço para alterações na implementação de um compilador. Durante o capítulo, são mostrados exemplos de alterações na linguagem *Small*, mostrando os impactos em sua implementação. Como validação, é destacada a experiência do uso do *ACOA* na implementação do compilador de Machĭna.

5.1 Facilidades de Uso

Facilidade de uso foi um dos requisitos impostos ao*ACOA*. Um arcabouço que traga dificuldades no seu uso causa problemas em sua aprendizagem e problemas nas alterações das aplicações, principalmente quando o usuário que estiver fazendo as alterações não seja o mesmo do desenvolvimento, mesmo que este arcabouço proporcione uma maior flexibilidade para as alterações.

Uma das facilidades do *ACOA* é a linguagem de especificação para o *FrEG*. Essa linguagem tem uma sintaxe semelhante à de outras linguagens de especificação para uma ferramenta *Compiler-Compiler*, como para o *Yacc* e o *Bison*, permitindo desse modo uma certa familiaridade por parte dos usuários que usam ou já usaram essas ferramentas. Porém, a linguagem para o *FrEG* é mais abstrata em relação à de outras ferramentas, limitando o tipo de ação semântica de uma produção da gramática a apenas à construção do nodo da AST, permitindo que o analisador sintático gerado construa toda AST. Isso faz com que o *FrEG* seja mais específico que as tradicionais ferramentas *Compiler-Compilers*, entretanto o *FrEG* permite maior produtividade ao usuário que busca apenas construir a AST como ação semântica

para o analisador sintático, além de permitir que o usuário se concentre mais nos detalhes léxicos e sintático de sua linguagem, deixando os detalhes de implementação dessas partes do compilador para o *FrEG*.

O arquivo de especificação para o *FrEG* possui importantes informações ao usuário para ser usadas na fase de implementação dos passos de compilação. Ao usar a inserção estática para implementar um dos passos de compilação, o usuário implementa esse método como se estivesse implementando qualquer método para uma classe, ou seja, o usuário pode usar qualquer membro ou método, sendo eles privados ou públicos, que a classe possuir. Portanto, a partir do arquivo de especificação, o usuário tem as informações sobre as classes dos nodos da AST, como nomes de membros privados, métodos de acesso aos membros privados, métodos de alteração do valor de um membro privado e os métodos que implementam os passos de compilação. Com isso, o usuário tem a vantagem de não precisar ler qualquer código gerado para conhecer as informações sobre as classes. As únicas informações que o usuário não pode extrair do arquivo de especificação para o *FrEG* são alguns métodos de auxílio que todas as classes da AST possuem, que são o método *_Signature* e o método *_getPosCodeSource*, que foram comentado no Capítulo 4.

A inserção estática de métodos e membros da programação orientada por aspectos é outra vantagem do *ACOA*. Além da inserção estática não possuir os problemas encontrados no padrão de projeto Visitor, a inserção estática mantém um alto grau de modularização, o que aumenta a extensibilidade do arcabouço, permitindo uma maior facilidade de alterações nas implementações dos compiladores. Outra vantagem da inserção estática é o fato de permitir que usuários que não possuem vastos conhecimentos sobre a programação orientada por aspectos possam usá-la, pois sua idéia é simples e sua sintaxe também.

O código gerado automaticamente e o código criado pelo usuário são totalmente integrados de modo automático pelo costurador de AspectC++ sem nenhuma necessidade do usuário fazer qualquer alteração no código gerado automaticamente.

É importante para o usuário ter entendimento completo do funcionamento do compilador implementado por ele, que detenha algum conhecimento dos requisitos de projeto como arquitetura, divisão de classes e fluxo de controle que são definidos pelo o arcabouço e que são reusados na implementação dos compiladores, promovendo deste modo o reúso de projeto. No entanto, esses requisitos são simples de serem entendidos, permitindo que o usuário tenha conhecimento teórico dessas partes, não havendo necessidade de conhecimento de implementação.

5.2 Recurso para Alterações

Esta seção mostra a flexibilidade do *ACOA* para alterações nas implementações dos compiladores desenvolvidos usando o arcabouço. São ilustrados alguns exemplos de alterações na linguagem *Small*, mostrando o que é necessário alterar e adicionar na implementação original.

As alterações foram divididas em categorias:

- alteração no número ou na ordem dos passos de compilação;
- adição ou remoção de produções da gramática;
- alterações de implementação dos passos de compilação.

5.2.1 Alteração no número ou na ordem dos passos de compilação

Esta seção demonstra as alterações relacionadas à adição, remoção e alteração na ordem de execução dos passos de compilação.

Suponha que deseje que o compilador da linguagem *Small* passe a gerar código em Pascal, além de continuar gerando código C. Para isso, basta adicionar um novo passo de compilação chamado *GeraCodigoPascal* no compilador. O arquivo de especificação para o *FrEG*, mostrado na Seção B.1 do Apêndice B, é alterado como mostrado abaixo, acrescentando o novo passo de compilação.

```
%%Passes
verificaNome verificaFunPro verificaTipo geraCodigoFunPro geraCodigo
GeraCodigoPascal
```

Após essa alteração da especificação, é necessário executar o *FrEG* e em seguida o usuário tem que implementar este novo passo para todas as classes concretas da AST. Essa alteração no compilador não altera qualquer implementação dos outros passos existentes.

Eventualmente, na adição de um novo passo, é necessário fornecer algumas informações no momento de sua execução. Se essas informações não foram recolhidas em um passo já existente, que execute antes do novo passo, o usuário tem a opção de alterar algum passo que execute antes do novo passo ou criar um outro passo para fornecer essas informações sem a necessidade de alterar outros passos.

Considere agora a alteração na ordem de execução dos passos de compilação. Considerando o exemplo anterior, o compilador de *Small* passa a gerar código Pascal antes de gerar código em C. Portanto, é necessário alterar

a ordem no arquivo de especificação, como mostrado a seguir, e em seguida executar o *FrEG*.

```
%%Passes
verificaNome verificaFunPro verificaTipo geraCodigoFunPro GeraCodigoPascal
geraCodigo
```

Para a remoção de um passo de compilação, basta retirar a declaração do passo do arquivo de especificação e em seguida executar o *FrEG*. Se deseja-se que o compilador de *Small* passe a gerar apenas código Pascal, deve-se remover o passo que gera código em C, como mostrado abaixo.

```
%%Passes
verificaNome verificaFunPro verificaTipo geraCodigoFunPro GeraCodigoPascal
```

A criação de um novo passo de compilação, ou 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, como pode ser observado nos exemplos.

5.2.2 Adição ou remoção de produções da gramática

Esta seção demonstra as alterações relacionadas à adição e remoção de produções da gramática. Os exemplos mostrados nesta seção também consideram a implementação do compilador de *Small*, mostrado no Apêndice B.

Inicialmente, é adicionando o novo comando *Do While* em *Small*, como mostrado na especificação do *FrEG*, abaixo:

```
...
Com:Comandos ::= Identificador Oatr Exp
          -> Atr(Identificador:id, Exp:Lado_dir)
        |  While Exp Do Com
          -> WhileComando(Exp:Condicao, Com:Com_While)
        |  If Exp Then Com1 Else Com2
          -> IfComando(Exp:Condicao, Com1:Com_Then, Com2:Com_Else)
        |  Do Com While Exp
          -> DoWhileComando(Com: Com_DWhile, Exp:Condicao)
        ;
...
```

Adição de produções na gramática como esta, que não altera a semântica de outras produções, gera apenas a necessidade de alterar a especificação para o *FrEG*, em seguida executar o *FrEG* e implementar todos os passos de compilação existentes para a nova classe ou as novas classes de nodos da AST que surgiram com a adição da nova ou novas produções. No caso do

comando *Do While*, surgiu apenas a classe *DoWhileComando*. A Figura 5.1 exemplifica a implementação para a verificação de tipo para o comando *Do While*.

```
...

aspect DoWhileComandoAsp {

public:
    ...

    advice "DoWhileComando" : void verificaTipo() {
      Condicao->verificaTipo();
      if(Condicao->gettipo() == boolean)
        Com_While->verificaTipo();
      else
        erros->push_back(...);
    }

    ...

};
```

Figura 5.1: DoWhileComandoAsp.ah

Considere agora a adição da produção que permite a passagem de parâmetros por referência em um procedimento de *Small*. As alterações para a especificação do *FrEG* são mostradas abaixo.

```
...
Decl:Declaracoes ::= ...
     | Proc Identificador1 left_parenthisis Type Identificador2
            right_parenthisis semicolon DeclList Com
         -> DeclProc(Identificador1:ProcNome, Identificador2:ParNome, Type:tp,
                     DeclList : Proc_Decl, Com :ProcCom)
     | Proc Identificador1 left_parenthisis Var Type Identificador2
            right_parenthisis semicolon DeclList Com
         -> DeclProcPaRef(Identificador1:ProcNome, Identificador2:ParNome, Type:tp,
                     DeclList : Proc_Decl, Com :ProcCom)
        ;
...
```

Esse tipo de produção altera a semântica de outras produções. Esse exemplo altera a semântica da chamada de procedimento, no qual, no ponto de chamada, a expressão passada para o parâmetro por referência tem que ser uma variável.

No caso do parâmetro ser por referência, o usuário tem que implementar os passos de compilação existentes para a nova classe da AST e tem que fazer outras alterações para verificar a expressão passada como parâmetro no caso do parâmetro ser por referência.

Uma opção é alterar, por exemplo, o passo de verificação de tipo para a classe *ChamaProc*. Outra opção é adicionar um novo passo de compilação que verifica se a expressão passada para um parâmetro por referência é uma variável. Neste caso, o usuário tem que implementar para todas as classes concretas da AST esse novo passo, mas não existe a necessidade de alterar qualquer outro passo. Na Figura 5.2, é mostrada a implementação desse novo passo, chamado de *VerificaParametroRef*, para todas as classes concretas da AST.

```
#ifndef _VERIFICAPAAMETROREFASP_AH_
#define _VERIFICAPAAMETROREFASP_AH_

...

aspect VerificaParametroRefAsp {

public:
pointcut chamadaDeProcedimento() = "ChamaProc";

advice !chamadaDeProcedimento() : void VerificaParametroRef() {}

advice chamadaDeProcedimento() : void VerificaParametroRef() {
  ItemFuncProc *ifp = (ItemFuncProc *)tab->busca(ProcNome->getToken_value());
  //verifica se o parâmetro é por referência
  if(ifp->getParametroReferencia()) {
    //verifica se a expressão é um identificado
    if(Parametro->_Signature() == "IdExp") {
      IdExp *ie = (IdExp *) Parametro;
      Item *it = tab->busca(id->getid()->getToken_value());
      if(it->gettipo_construcao() != var) //verifica se é uma variável
         erros->push_back(...);
    }
    else
     erros->push_back(...);
  }
}
};

#endif
```

Figura 5.2: VerificaParametroRefAsp.ah

Observando a implementação acima, todas as classes concretas da AST, com exceção da classe *ChamaProc*, possuem a mesma implementação. Esta é outra vantagem da programação orientada por aspectos, que permite evitar repetições de códigos.

Nem todas as novas produções que alteram semanticamente outras produções

ocasionam o surgimento de um novo passo de compilação para a implementação da verificação de sua semântica. No caso da verificação da passagem por referência isso foi possível devido ao fato de haver a necessidade de adicionar uma verificação que antes não existia.

Suponha agora que é adicionado o tipo *Float* na linguagem *Small*, como mostrado abaixo:

```
...
Type  ::= Int -> Inteiro()
       |  Bool -> Booleano()
       |  Float -> Real()
        ;
...
```

Essa produção altera a semântica das operações binárias de adição, subtração, multiplicação e divisão. Essas operações originalmente são realizadas usando apenas operandos do tipo *Int*. Com a adição do tipo *Float*, as operações podem ser realizadas com ambos os tipos, porém os operandos têm que possuir o mesmo tipo. A Figura 5.3 mostra a nova implementação do passo de verificação de tipo para essas operações.

```
#ifndef _EXPRESSOESASP_AH_
#define _EXPRESSOESASP_AH_
...
aspect ExpressoesAsp {
...
public:
    ...
    advice "Adicao" || "Subtracao" || "Multiplicacao"
       || "Divisao" : void verificaTipo() {
        Exp1->verificaTipo();
        Exp2->verificaTipo();
        //verifica se o primeiro operando é do tipo inteiro ou real
        if(Exp1->gettipo() != integer || Exp1->gettipo() != real) {
          erros->push_back(...);
          return;
        }
        //verifica se o segundo operando possui o mesmo tipo do primeiro
        if(Exp1->gettipo() != Exp2->gettipo()) {
          erros->push_back(...);
          return;
        }
        tipo = Exp1->gettipo(); //tipo da expressão
    }
    ...
};

#endif
```

Figura 5.3: ExpressionAsp.ah

Nesse caso, não foi possível evitar alteração de código já existente porém, como o *ACOA* é bastante modularizado, é possível saber com exatidão os locais que serão alterados.

A remoção de produções da gramática segue o mesmo raciocínio da adição. A remoção de produções que não alteram semanticamente outras produções faz com que não exista a necessidade de qualquer alteração em outras partes do compilador. Em casos onde as produções alterem outras produções semanticamente, vão ser necessários também alterações em outras partes do compilador.

5.2.3 Alterações de implementação dos passos de compilação

Esta seção demonstra as alterações relacionadas a mudanças de implementação dos passos de compilação já existentes, caso a linguagem sofra alterações semânticas.

Para exemplificar, considere a semântica da operação binária de adição, subtração, multiplicação e divisão mostrada na Seção 5.2.2. Alterando a semântica dessas operações para permitir que os operandos possuam tipos diferentes, a Figura 5.4 mostra a nova implementação para a verificação de tipo para estas operações.

Neste caso, também não é possível evitar alterações de código existentes mas, pela modularização, não é necessário alterar implementação de outros passos.

5.3 Avaliação Qualitativa

Esta seção compara, de modo qualitativo, o custo de construção de um compilador usando o *ACOA* com o esforço necessário para de construção de um compilador de forma manual.

Supondo a construção de dois compiladores, um deles construído usando o *ACOA* e outro manualmente. O compilador construído usando o *ACOA* é chamado de *C* enquanto o compilador construído manualmente de *C'*. Os compiladores *C* e *C'* são idênticos, possuindo as mesmas classes, apenas o modo de construir os compiladores é que difere.

O compilador *C* é construído com o *ACOA* usando os passos descritos anteriormente. O compilador *C'* é construído do seguinte modo:

- o analisador léxico é construído usando a ferramenta *Flex*.
- O analisador sintático é construído usando a ferramenta *Bison*.

```
#ifndef _EXPRESSOESASP_AH_
#define _EXPRESSOESASP_AH_
...
aspect ExpressoesAsp {
...
public:
    ...
    advice "Adicao" || "Subtracao" || "Multiplicacao"
       || "Divisao" : void verificaTipo() {
        Exp1->verificaTipo();
        Exp2->verificaTipo();
        //verifica se o primeiro operando é do tipo inteiro ou real
        if(Exp1->gettipo() != integer || Exp1->gettipo() != real) {
          erros->push_back(...);
          return;
        }
        //verifica se o segundo operando é do tipo inteiro ou real
        if(Exp2->gettipo() != integer || Exp2->gettipo() != real) {
          erros->push_back(...);
          return;
        }
        if(Exp1->gettipo() == real)
          tipo = Exp1->gettipo(); //tipo da expressão
        else
          if(Exp2->gettipo() == real)
            tipo = Exp2->gettipo();
          else
            tipo = integer;

    }
    ...
};

#endif
```

Figura 5.4: ExpressionAsp.ah alterada

- As classes de nodos da AST e qualquer outra parte gerada automaticamente pelo *ACOA* são construídas manualmente.
- A implementação de novos passos de compilação é feita em métodos das classes dos nodos da AST. Neste caso, estes métodos são implementados diretamente nas classes, violando a encapsulação.
- Outros membros e métodos inseridos estaticamente nas classes dos nodos da AST na implementação usando o *ACOA* são implementados diretamente nas classes.

A Tabela 5.2 compara a complexidade de implementação de partes do compilador de *C* e *C'*.

A construção dos analisadores léxicos e sintáticos possui menor complexidade com a utilização do *ACOA*, pois a linguagem de especificação para o *FrEG* é mais abstrata que a linguagem do *Flex* e *Bison*. O uso do *ACOA*

	C	*C'*
analisador léxico	menor	maior
analisador sintático	menor	maior
classes de nodos da AST	menor	maior
passos de compilação	igual	igual

Tabela 5.1: Comparação da complexidade da implementação de partes dos compiladores

automatiza o processo de implementação das classes de nodos da AST, o que torna a atividade menos complexa. As complexidades de implementação dos passos de compilação são iguais para ambos os métodos, pois a dificuldade de implementar um método de uma classe usando a inserção estática ou implementando diretamente na classe é a mesma.

A Tabela 5.2 compara a complexidade de fazer alterações nas implementações dos compiladores *C* e *C'*.

	C	*C'*
adicionar novos passos de compilação	menor	maior
remover passos de compilação	igual	igual
alterar a ordem de execução dos passos de compilação	igual	igual
adicionar novas produções na gramática	menor	maior
remover produções na gramática	igual	igual
alterar implementações dos passos de compilação	igual	igual

Tabela 5.2: Comparação da complexidade de alteração na implementação dos compiladores

A complexidade de se alterarem as implementações dos passos de compilação são iguais, pois a implementação de *C* e de *C'* possuem o mesmo nível de modularização. A remoção de produções gramaticais possui o mesmo nível de complexidade para ambas as implementações, pois se existir a necessidade de alterar a semântica de outras produções será necessário, portanto alterar a implementação de algum passo de compilação. A remoção ou alteração na ordem de execução dos passos de compilação possui complexidade igual, pois essas alterações não possuem significativas alterações em ambos os casos.

A adição de novas produções da gramática tem complexidade menor para a implementação com o *ACOA*, pois este tipo de alteração acarreta na criação de novas classes da AST o que obriga, na implementação manual, a criação de novas classes pelo desenvolvedor.

A diferença mais significativa nessa comparação é a adição de novos passos

de compilação. No caso de novos passos para o compilador C', é necessário alterar as classes concretas dos nodos da AST, adicionando um novo método para cada classe. Com o uso do *ACOA*, isso é feito de modo simples, pela inserção estática da programação orientada por aspectos.

5.4 Compilador Machĭna

Para uma validação mais substantiva de *ACOA*, o arcabouço foi usado para a implementação de um compilador para a linguagem Machĭna, que foi apresentada na Seção 1.2. O *ACOA* faz parte do Projeto Machĭna, tendo como objetivo a compilação de código escrito em Machĭna para um código em MIR.

Machĭna é uma linguagem bastante 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, ou seja, classes concretas para os terminais, classes abstratas para os não-terminais e classes concretas para as produções gramaticais.

Foram implementados 4 passos de compilação para o compilador de Machĭna, chamados de: *IncluiModuloInterface*, *MontaTabelaGlobal*, *VerificaTipo* e *GeraCodigo*.

O passo *IncluiModuloInterface* busca, no diretório corrente do módulo que está sendo compilado, outros módulos no qual seu vocabulário é incluído no módulo compilado. Para o funcionamento dessa busca, os arquivos que implementam um módulo devem possuir o mesmo nome do módulo, com a terminação *.mc*. Portanto, um módulo de nome *M* é implementado em um arquivo com o nome de *M.mc*. Depois da busca, o compilador de Machĭna cadastra, na tabela de símbolos do módulo que está sendo compilado, os símbolos públicos do módulo incluído.

Todas as dependências dos módulos incluídos devem aparecer no módulo compilado. Por exemplo, o módulo *M* inclui o módulo *M2*, porém *M2* necessita de definições contidas no módulo *M1*, portanto em *M* deve existir, além da inclusão de *M2*, a inclusão de *M1*. Abaixo, pode ser observado o código em Machĭna para essa inclusão:

```
module M

  include M1, M2;
  ...
```

```
end M
```

Observando o código, *M1* tem que ser incluído antes de *M2*, para que suas definições estejam disponíveis antes do compilador tratar *M2*.

Nenhum módulo incluído pode depender de definições do módulo que os inclui, pois isso causaria uma dependência circular. Portanto, no exemplo, os módulos *M1* e *M2* não podem ter seu vocabulário público dependente de definições de *M*.

Outra funcionalidade do passo *IncluiModuloInterface* é a busca de interfaces, que permite que um agente do módulo compilado comunique com outros agentes. Para o funcionamento dessa busca, os arquivos que implementam uma interface devem possuir o mesmo nome do módulo que a interface representa, com a terminação *.itf*. Portanto, o arquivo de implementação de uma interface para o módulo *M* tem o nome de *M.itf*. Depois da busca, o compilador inclui na tabela de símbolos os símbolos da interface.

O passo *MontaTabelaGlobal* insere os símbolos públicos e privados do módulo que está sendo compilado, na tabela de símbolos. Nesse passo, os símbolos incluídos por outros módulos estão disponíveis para serem usados.

O passo *VerificaTipo* verifica se os símbolos usados nas regras de transição foram declarados. Esse passo também faz a verificação de tipos necessários nas regras de transição, na inicialização de funções e na declaração de um valor *default* para os tipos.

O passo *GeraCodigo* transcreve código escrito em Machĭna em código MIR.

O compilador de Machĭna foi chamado de *machinac* para ativá-lo basta chamá-lo em um terminal, passando os arquivos fontes, como mostrado abaixo:

```
machinac arquivos_fonte.mc {arquivos_fonte.mc}
```

Sua saída é um arquivo fonte escrito em código MIR, para cada arquivo passado para o compilador. O arquivo com definição em MIR tem terminação *.mod* ou *.mas*. O *.mod* é o arquivo escrito em MIR que possui a definição de um módulo de Machĭna, e o arquivo *.mas* é um arquivo escrito em MIR que possui a definição de uma máquina de Machĭna.

A Seção 5.3 faz a comparação qualitativa da complexidade de implementação e da complexidade de alteração da implementação de um compilador usando o *ACOA* e outro criado de modo manual.

Suponha agora,que um compilador *machinac'* tenha sido implementado manualmente utilizando os mesmos critérios descritos na Seção 5.3. A Tabela 5.3 compara quantitativamente as duas implementações, considerando como parâmetro o número aproximado de linhas de código criadas pelo usuário.

	machinac	*machinac'*
linhas de código	25000	61000

Tabela 5.3: Número aproximados de linhas de código

Observando o resultado, as 36000 linhas de código que *machinac'* teria a mais, são exatamente o número de linhas de código que o *ACOA* gera automaticamente para *machinac*. Nesse caso, cerca de 59% das linhas de código do compilador *machinac* foram geradas automaticamente pelo *ACOA*.

Portanto, os compiladores desenvolvidos usando o *ACOA*, além de ter uma complexidade de implementação e alteração menor do que um compilador igualmente criado de forma manual, proporcionam ao usuário um ganho de desempenho no seu desenvolvimento, devido à grande quantidade de código criado automaticamente.

5.5 Conclusão

Este capítulo mostrou as facilidades do uso do *ACOA* e também mostrou a importância da modularização, o que permite maior facilidade e flexibilidade para realizações de alterações nas implementações dos compiladores.

O compilador de Machĭna enfatiza algumas características do *ACOA* para a facilidade de seu desenvolvimento e manutenção. Linguagens extensas como Machĭna deixam a criação das classes de nodos da AST de modo manual praticamente inviável, e sem o uso de uma AST a modularização do compilador seria menor, o que tornaria o compilador mais complexo de ser alterado e, portanto menos extensível.

Outra boa característica do *ACOA* para o desenvolvimento de compiladores para linguagens extensas como Machĭna é a possibilidade de criar uma única implementação de um método para várias classes da AST, via inserção estática. Muitas classes não participam de algum passo de compilação, por exemplo, as classes relacionadas às regras de transição de Machĭna não participam dos passos *IncluiModuloInterface* e *MontaTabelaGlobal*, permitindo ao usuário desenvolver apenas uma única implementação para essas classes o que permite que ele concentre mais suas atenções nas classes que precisam de uma implementação específica.

Para manutenção de compiladores de linguagens grandes como Machĭna, a flexibilidade do *ACOA* para alterações nas implementações dos compiladores é uma característica importante. Isto permite que o usuário muitas vezes não altere códigos existentes ou, quando necessitar, será fácil saber os locais exatos das alterações.

Capítulo 6

Conclusão e Trabalhos Futuros

O objetivo de construir um arcabouço que seja simples de ser usado para o desenvolvimento de compiladores e permitir que futuras alterações sintática e semântica na linguagem possam ser de fácil alteração na implementação do compilador foi alcançado com o arcabouço *ACOA*.

O *ACOA* alcança uma facilidade de desenvolvimento pelo fato de automatizar etapas desse desenvolvimento e pelo uso da inserção estática nas etapas que são construídas pelo usuário. O *ACOA* alcança uma facilidade de alterações nas implementações dos compiladores devido ao fato de ser bastante modularizado, o que lhe permite ser bastante flexível para as realizações das alterações.

O *ACOA* é um arcabouço para a construção de linguagens em geral, gerando compiladores escritos em C++. Sua utilização em um ambiente ASM, como no *Projeto Machĭna*, pode gerar contribuições interessantes pois como o modelo ASM é um campo ainda em aberto, havendo vários recursos não explorados, o *ACOA* permite que os compiladores construídos para essas linguagens possam sofre muitas alterações.

6.1 Principais Contribuições

As principais contribuições alcançadas com esta dissertação foram:

O arcabouço *ACOA*: O *ACOA* é um arcabouço para implementações de compiladores de fácil utilização, flexível nos recursos de alterações das implementações dos compiladores desenvolvidos e permite ao usuário um ganho de desempenho devido a várias partes que são geradas automaticamente. Outra contribuição do *ACOA* é a ferramenta *FrEG* e sua linguagem de especificação. Essa linguagem é simples e bastante

abstrata, o que permite que o usuário concentre a maior parte de suas atenções nos detalhes léxico e gramatical de sua linguagem.

O compilador de Machĭna: A implementação do compilador de Machĭna usando o *ACOA* permite que trabalhos futuros explorem a linguagem Machĭna, permitindo uma maior facilidade nas alterações na implementação do compilador

Aplicação da AOP na construção de compiladores: A programação orientada por aspectos é uma tecnologia nova na ciência da computação, que ainda pode ser bastante explorada. Na área de compiladores, existem poucos trabalhos relacionados ao uso da programação orientada por aspectos. Esta dissertação aplica a programação orientada por aspectos na construção de compiladores, pretendendo contribuir para futuras explorações do uso da programação orientada por aspectos na construção de compiladores.

6.2 Trabalhos Futuros

O desenvolvimento do compilador de Machĭna usando o arcabouço ACOA abre possibilidade para facilitar alterações nesse compilador e de evoluções na linguagem. Algumas dessas possibilidades são mostradas abaixo:

Alteração da geração de código: a implementação do compilador Machĭna gera arquivos contendo código MIR com a sintaxe de XML, o qual é a entrada para o arcabouço kℓar , para posterior geração de código C++. É possível alterar o compilador de Machĭna para que possa usar a biblioteca do arcabouço kℓar possibilitando assim que a representação de MIR fique na memória obtendo desse modo maior integração entre o compilador de Machĭna e o kℓar .

Outra possibilidade de alteração na geração de código do compilador de Machĭna é permitir a geração de código diretamente em C++.

Construção do compilador para a linguagem AspectM: a linguagem *AspectM* é uma extensão da linguagem Machĭna, adicionando características de orientação por aspectos. Essa linguagem está sendo proposta em uma dissertação de mestrado que está em andamento. O compilador de *ApectM* pode utilizar o *ACOA* para alterar a implementação original do compilador de Machĭna.

Apêndice A

Gramática da Entrada do *FrEG*

```
Start_Grammar       ::= Lexer_Def Parser_Def Passes_Def
Lexer_Def           ::= "%%Lexer" Lexer_Rules
Lexer_Rules         ::= {Synonymous} [Conditions] [Patterns]
Synonymous          ::= Name Regular_Expression {Regular_Expression}
Name                ::= Identifier
Regular_Expression  ::= String
                     |  "[" {Letter | Digit | Symbol} "]"
                     |  "."
                     |  Regular_Expression "*"
                     |  Regular_Expression "+"
                     |  Regular_Expression "?"
                     |  Regular_Expression
                        "{"number ( ","] | ["," number] )"}"
                     |  "{" Name "}"
                     |  "\0"
                     |  "\123"
                     |  "\x2a"
                     |  "(" Regular_Expression ")"
                     |  Regular_Expression "|" Regular_Expression
                     |  Regular_Expression "/" Regular_Expression
                     |  "^" Regular_Expression
                     |  Regular_Expression "$"
                     |  "<" Name ">" Regular_Expression
                     |  "<" "*" ">" Regular_Expression
                     |  "<<" "EOF" ">>"
Conditions          ::= "%Conditions" Conditions_list
Conditions_list     ::= Conditions_mode {Conditions_mode}
Conditions_mode     ::= ("%inclusive" | "%exclusive") Name {Name}
Patterns            ::= "%Pattern" Pattern {Pattern}
```

```
Pattern             ::= Token_Def [Action] ";"
Token_Def           ::= Regular_Expression {Regular_Expression}
Action              ::= "->" Begin_End_Conditons ["," EndLine]
                              ["," Name_Token_AST]
                      |  "->" EndLine ["," Begin_End_Conditons]
                              ["," Name_Token_AST]
                      |  "->" Name_Token_AST
                      |  "->" Error
Begin_End_Conditons ::= "BEGIN" "(" Name ")"
                      | "END" "(" ")"
EndLine             ::= "endline"
Error               ::= "ERROR"
Name_Token_AST      ::= Token_Type
Token_Type          ::= Terminal
Parser_Def          ::= "%%Parser" Parser_Rules
Parser_Rules        ::= Declaration [Grammar]
Grammar             ::= "%Grammar" {Rule}
Declaration         ::= {Precedence} [Start]
Precedence          ::= ("%left" | "%right" | "%nonassoc")
                        (Terminal | Name) {Terminal | Name}
Start               ::= "%start" Nonterminal
Rule                ::= Left_Hand "::=" Right_Hand {"|" Right_Hand} ";"
Left_Hand           ::= Tagged_Nonterminal
Tagged_Nonterminal  ::= Nonterminal [":"Ast_Id]
Right_Hand          ::= Production_Elements [ "->" Ast_Construction ]
Production_Elements ::= {Nonterminal | Terminal}
                      |  {Nonterminal | Terminal}
                        "%prec" (Name | Terminal)
Ast_Construction    ::= Ast_Id "(" [Ast_Elements] ")"
Ast_Id              ::= Identifier
Ast_Elements        ::= Ast_Element {"," Ast_Elements }
Ast_Element         ::= Tagged_Nonterminal | Tagged_Terminal
Tagged_Terminal     ::= Terminal[":"Ast_Id]
Passes_Def          ::= "%%Passes" {Pass}
Pass                ::= Name_Pass
Name_Pass           ::= Identifier
Terminal            ::= Identifier
Nonterminal         ::= Identifier
Identifier          ::= letter { "_" | Letter | Digit}
String              ::= """ {Letter | Digit | Symbol} """
Number              ::= Digit {Digit}
Digit               ::= "0" | "1" | "2" | "3" | "4"
                      |  "5" | "6" | "7" | "8" | "9"
```

```
Letter              ::= "a" | "b" | "c" | "d" | "e"
                      |  "f" | "g" | "h" | "i" | "j" |
                      |  "k" | "l" | "m" | "n" | "o"
                      |  "p" | "q" | "r" | "s" | "t"
                      |  "u" | "v" | "w" | "x" | "y"
                      |  "z" | "A" | "B" | "C" | "D"
                      |  "E" | "F" | "G" | "H" | "I"
                      |  "J" | "K" | "L" | "M" | "N"
                      |  "O" | "P" | "Q" | "R" | "S"
                      |  "T" | "U" | "V" | "W" | "X"
                      |  "Y" | "Z"
Symbol              ::= """ | "\" | "[" | "]" | "^" | "-" | "?"
                      |  "." | "*" | "+" | "|" | "(" | ")" | "$"
                      |  "/" | "{" | "}" | "%" | "<" | ">" | "!"
                      |  "@" | "#" | "&" | "_" | "=" | "," | ";"
                      |  ":" | "~" | "\"
```

Apêndice B

Compilador de Small

Este apêndice apresenta a implementação de um compilador para *Small* usando o *ACOA*. O compilador implementado faz as seguintes análises semânticas e geração de código: verificação de nomes (*verificaNome*), verificação de nomes e tipos em funções e procedimentos (*verificaFunPro*), verificação de tipos (*verificaTipo*), gera código para funções e procedimentos (*geraCodigoFunPro*) e gera código para o programa (*geraCodigo*). A geração de código gera código C++ como código intermediário.

- As seções desse apêndice mostram as seguintes partes da implementação:
- Seção B.1 definição de *Small* pra o *FrEG*.
- Seção B.2 implementação da tabela de símbolos.
- Seção B.3 implementação da função *main*.
- Seção B.4 implementação das análises semânticas e geração de código através dos aspectos.

B.1 Definição de *Small* para o *FrEG*

```
%%Lexer

letras [A-Za-z_]
digitos [0-9]
id {letras}({letras}|{digitos})*
numero {digitos}+


%Conditions
```

```
%inclusive comentarios


%Pattern


"//"[^\n]*                          ;
"/*"                                    -> BEGIN(comentarios);
<comentarios>"*/"                   -> END();
<comentarios>"\n"                   -> endline;
<comentarios>[^*\n]*            ;
<comentarios>"*"+[^*/\n]*       ;

{numero}                        -> ConstanteInt ;
"program"                       -> program;
"if"                            -> If;
"then"                          -> Then;
"else"                          -> Else;
"true"                          -> True;
"false"                         -> False;
"Begin"                         -> Begin;
"End"                           -> End;
"Int"                           -> Int;
"Bool"                          -> Bool;
"Output"                        -> Output;
"while"                         -> While;
"do"                            -> Do;
"not"                           -> Not;
"Read"                          -> Read;
"var"                           -> Var;
"const"                         -> Const;
"func"                          -> Func;
"proc"                          -> Proc;
"and"                           -> andop;
{id}                            -> Identificador;
":="                            -> Oatr;
";"                             -> semicolon;
","                             -> comma;
"+"                             -> add;
"-"                             -> sub;
"*"                             -> mult;
"<"                             -> menor;
"("                             -> left_parenthisis;
")"                             -> right_parenthisis;
"="                             -> igual;
":"                             -> colon;
"/"                             -> divi;
[\n]                            -> endline;
[ \t\r]                         ;
.                               -> ERROR;


%%Parser

%left andop
%left menor igual
%left add sub
%left mult divi
%left Not

%start Program
```

```
%Grammar


Program  ::= program DeclList Com -> Inicia(DeclList:prog_declaracoes, Com : corpo) ;


DeclList ::= Decl semicolon DeclList -> SeqDecl(Decl : declaracao, DeclList : continuacao)
          | -> FimDeclaracao() ;


Decl:Declaracoes ::= Var Type Identificador igual Exp
                       -> DeclVar(Type:tp, Identificador:id, Exp:exp_inicial)

                   | Const Type Identificador igual Exp
                       -> DeclConst(Type:tp, Identificador:id, Exp:exp_inicial)

                   | Func Identificador1 left_parenthisis Type1 Identificador2
                     right_parenthisis colon Type2 semicolon DeclList Begin Exp End
                       -> DeclFunc(Identificador1:FuncNome, Type2,
                                   Identificador2:ParNome, Type1, DeclList : Func_Decl,
                                   Exp :FuncExp)

                   | Proc Identificador1 left_parenthisis Type Identificador2
                     right_parenthisis semicolon DeclList Com
                       -> DeclProc(Identificador1:ProcNome, Identificador2:ParNome, Type:tp,
                                   DeclList : Proc_Decl, Com :ProcCom)
                   ;


Type  ::= Int -> Inteiro()
       |  Bool -> Booleano();

ComList ::= Com semicolon ComList -> SeqCom(Com:comando, ComList : continuacao)
          | ;


Com:Comandos ::= Identificador Oatr Exp -> Atr(Identificador:id, Exp:Lado_dir)

              |  If Exp Then Com1 Else Com2
                       -> IfComando(Exp:Condicao, Com1:Com_Then, Com2:Com_Else)

              |  While Exp Do Com -> WhileComando(Exp:Condicao, Com:Com_While)

              |  Output left_parenthisis Exp right_parenthisis
                       -> OutputComando(Exp:saidaExp)

              |  Begin ComList End -> Bloco(ComList : lista_comandos)

              |  Identificador left_parenthisis Exp right_parenthisis
                       -> ChamaProc(Identificador:ProcNome, Exp:Parametro) ;


Exp:Expressoes ::=  ConstanteInt -> Numero(ConstanteInt)

                |   True -> ConstanteTrue(True)

                |   False -> ConstanteFalse(False)

                |   Exp1 add Exp2 -> Adicao(Exp1, Exp2)

                |   Exp1 sub Exp2 -> Subtracao(Exp1, Exp2)
```

```
|   Exp1 mult Exp2 -> Multiplicacao(Exp1, Exp2)

|   Exp1 divi Exp2 -> Divisao(Exp1, Exp2)

|   Exp1 menor Exp2 -> MenorExp(Exp1, Exp2)

|   Exp1 igual Exp2 -> IgualExp(Exp1, Exp2)

|   Exp1 andop Exp2   -> AndExp(Exp1, Exp2)

|   Not Exp        -> Negacao(Exp)

|   Identificador -> IdExp(Identificador:id)

|   left_parenthisis Exp right_parenthisis -> ParExp(Exp)

|   Identificador left_parenthisis Exp right_parenthisis
        -> ChamaFunc(Identificador:FuncNome, Exp:Parametro)

|   Read   -> ReadValor()

|   If Exp1 Then Exp2 Else Exp3 End
        -> IfExpressao(Exp1 : Condicao, Exp2 : Exp_Then, Exp3 : Exp_Else)

;

%%Passes

verificaNome verificaFunPro verificaTipo geraCodigoFunPro geraCodigo
```

B.2 Tabela de Símbolos

A tabela de símbolo foi implementada na classe *Tabela*. Ela armazena ponteiros da classe *Item* ou de sua subclasse *ItemFuncProc*.

A classe *Item* possui as seguintes informações:

- *nome*: o símbolo usado no código fonte.
- *tipo_construcao*: retorna o tipo da construção do símbolo. As construções são: variável (*val*), constante (*const*), função (*func*), procedimento (*proc*).
- *tipo_valor*: retorna o tipo do símbolo. Os tipos são: inteiro (*integer*), booleano (*boolean*) ou nenhum (*nothing*).

A classe *ItemFuncProc* representa as funções e procedimentos. Esta classe além das informações contidas em *Item* possui informações do tipo (inteiro ou booleano) do parâmetro da função ou do procedimento.

Tabela.h

```
#ifndef TABELA_H
#define TABELA_H

#include <vector>
#include "Item.h"

using namespace std;

class Tabela{

private:
  vector <Item *> *tabela_nomes;

public:
        Tabela();
        virtual ~Tabela();
        bool insere(Item *);
        Item* busca(string );
        Item* busca(char *);

};

#endif // TABELA_H
```

Tabela.cpp

```
#include "Tabela.h"

Tabela::Tabela() {
  tabela_nomes = new vector<Item *>();
}

Tabela::~Tabela(){
  delete tabela_nomes;
}

bool Tabela::insere(Item *it) {
        for(int i = 0; i < tabela_nomes->size(); i++)
          if(it->getnome() == (*tabela_nomes)[i]->getnome())
            return false;

        tabela_nomes->push_back(it);

        return true;

}

Item* Tabela::busca(string n) {
        for(int i = 0; i < tabela_nomes->size(); i++)
          if(n == (*tabela_nomes)[i]->getnome())
            return (*tabela_nomes)[i];

        return NULL;

}

Item* Tabela::busca(char *n) {
        string temp = n;
```

```
        for(int i = 0; i < tabela_nomes->size(); i++)
          if(temp == (*tabela_nomes)[i]->getnome())
            return (*tabela_nomes)[i];

        return NULL;

}
```

Item.h

```
#ifndef ITEM_H
#define ITEM_H

#include <string>

using namespace std;

enum tipos_valor {boolean, integer, nothing};
enum tipos_construcao{var, constante, function, procedure};

class Item{

protected:
    string *nome;
    int tipo_construcao;
    int tipo_valor;

public:
        Item(string , int, int);
        Item(char *, int, int);
        virtual ~Item();
        string getnome();
        int gettipo_valor();
        int gettipo_construcao();
        void settipo_valor(int );

};

#endif // ITEM_H
```

Item.cpp

```
#include "Item.h"

Item::Item(string n, int tc, int tv) {
  nome = new string(n);
  tipo_construcao = tc;
  tipo_valor = tv;
}

Item::Item(char *n, int tc, int tv) {
  nome = new string(n);
  tipo_construcao = tc;
  tipo_valor = tv;
}

Item::~Item() {
  delete nome;
}
```

```
string Item::getnome() {
        return *nome;
}

int Item::gettipo_valor() {
        return tipo_valor;
}

int Item::gettipo_construcao() {
        return tipo_construcao;
}

void Item::settipo_valor(int t) {
        tipo_valor = t;
}
```

ItemFuncProc.h

```
#ifndef ITEMFUNCPROC_H
#define ITEMFUNCPROC_H

#include "Item.h"

class ItemFuncProc : public Item {

private :
    int tipoParametro;

public:
        ItemFuncProc(string , int, int, int);
        ItemFuncProc(char * , int, int, int);
        virtual ~ItemFuncProc();
        int getTipoParametro();
};

#endif // ITEMFUNCPROC_H
```

ItemFuncProc.cpp

```
#include "ItemFuncProc.h"

ItemFuncProc::ItemFuncProc(string n, int tc, int tv, int tp)
  : Item(n, tc, tv) {
  nome = new string(n);
  tipoParametro = tp;
}

ItemFuncProc::ItemFuncProc(char *n, int tc, int tv, int tp)
  : Item(n, tc, tv) {
  tipoParametro = tp;
}

ItemFuncProc::~ItemFuncProc() {
  delete nome;
}

int ItemFuncProc::getTipoParametro() {
        return tipoParametro;
}
```

B.3 Função Main

A função main usada foi a gerada pelo *FrEG*. O *FrEG* gera dois arquivos, o *main.cpp* e o *Global.h*. O arquivo *main.cpp* não sofreu alterações, foram adicionados os seguintes objetos de escopo global:

- *tab*: tabela de símbolos;
- *erros*: vetor que contém mensagens de erros;
- *saida*: ofstream que manipula o arquivo de saída onde o código gerado pelo compilador será salvo.

Global.h

```
#ifndef GLOBAL_H
#define GLOBAL_H

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

Tabela *tab;
vector<string *> *erros;
ofstream *saida;

#endif
```

main.cpp

```
#include <cstdio>
#include <iostream>
#include "Ast/Asts.h"
#include "Walk.h"
#include "Factory.h"
#include "Global.h"

using namespace std;

int yyparse();
int yylex();
extern FILE *yyin;
extern int yynum_erros;
Node *yyraiz;
Factory *yyFactory;

int main(int argc, char *argv[]) {

    Walk *w = new Walk();

    yyFactory = new Factory();
```

```
    yyin = fopen(argv[1], "r");

    if(yyin == NULL) {
        cout <<"Arquivo não existe." <<endl;
        exit(1);
    }

    yyparse();

    fclose(yyin);

    if(yynum_erros == 0)
        w->start(yyraiz);


    delete w;
    delete yyraiz;

    return 0;

}
```

B.4 Análise Semântica e Geração de Código

Nesta seção são mostrados os vários arquivos usados na implementação da análise semântica e da geração de código.

IniciaAsp.ah

```
#ifndef _INICIAASP_AH_
#define _INICIAASP_AH_

#include <iostream>
#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <cstdlib>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect IniciaAsp {

public:
    advice "Inicia" : void verificaNome() {
      tab = new Tabela();
      erros = new vector<string*>();
      prog_declaracoes->verificaNome();
      corpo->verificaNome();
      if(erros->size() > 0) {
        for(int i = 0; i < erros->size(); i++)
          cout <<*(*erros)[i] <<endl;
        exit(1);
```

```
    }
  }

  advice "Inicia" : void verificaFunPro() {
      prog_declaracoes->verificaFunPro();
      if(erros->size() > 0) {
        for(int i = 0; i < erros->size(); i++)
          cout <<*(*erros)[i] <<endl;
        exit(1);
      }
  }


  advice "Inicia" : void verificaTipo() {
      prog_declaracoes->verificaTipo();
      corpo->verificaTipo();
      if(erros->size() > 0) {
        for(int i = 0; i < erros->size(); i++)
          cout <<*(*erros)[i] <<endl;
        exit(1);
      }
  }

  advice "Inicia" : void geraCodigoFunPro() {
      saida = new ofstream("saida.cpp");
      if(!saida) {
              cout <<"Erro ao abrir o arquivo saida" <<endl;
              exit(1);
      }
      (*saida) <<"#include <iostream>\n";
      (*saida) <<"using namespace std;\n\n";

      (*saida) <<"int Read() {\n";
      (*saida) <<"  int temp;\n";
      (*saida) <<"  cin >> temp;\n";
      (*saida) <<"  return temp;\n";
      (*saida) <<"}\n\n";

      prog_declaracoes->geraCodigoFunPro();
  }


  advice "Inicia" : void geraCodigo() {
      (*saida) <<"int main() {\n\n";
      prog_declaracoes->geraCodigo();
      (*saida) <<"\n\n";
      corpo->geraCodigo();
      (*saida) <<"  return 0;\n";
      (*saida) <<"}\n";
      saida->close();
      delete tab;
      delete erros;
      delete saida;
  }

};

#endif
```

DeclListAsp.ah

```
#ifndef _DECLLIST_AH_
#define _DECLLIST_AH_

aspect DeclListAsp {

public:
    advice "DeclList" : virtual bool PossuiFunPro() = 0;
};

#endif
```

SeqDeclAsp.ah

```
#ifndef _SEQDECLASP_AH_
#define _SEQDECLASP_AH_

aspect SeqDeclAsp {

private:
    advice "SeqDecl" : bool FunPro;

public:
    advice "SeqDecl" : void verificaNome() {
      declaracao->verificaNome();
      continuacao->verificaNome();

      if(declaracao->_Signature() == "DeclFunc" ||
         declaracao->_Signature() == "DeclProc")
        FunPro = true;
      else
        if(continuacao->PossuiFunPro())
          FunPro = true;
        else
          FunPro = false;

    }

    advice "SeqDecl" : void verificaFunPro() {
        declaracao->verificaFunPro();
        continuacao->verificaFunPro();
    }

    advice "SeqDecl" : void verificaTipo() {
        declaracao->verificaTipo();
        continuacao->verificaTipo();
    }

    advice "SeqDecl" : void geraCodigo() {
        declaracao->geraCodigo();
        continuacao->geraCodigo();
    }

    advice "SeqDecl" : bool PossuiFunPro() {
        return FunPro;
    }

    advice "SeqDecl" : void geraCodigoFunPro() {
        declaracao->geraCodigoFunPro();
        continuacao->geraCodigoFunPro();
    }
```

```
};

#endif
```

FimDeclaracaoAsp.ah

```
#ifndef _FIMDECLARACAO_AH_
#define _FIMDECLARACAO_AH_


aspect FimDeclaracaoAsp {
private:
      advice "FimDeclaracao" : bool FunPro;

public:

   advice "FimDeclaracao" : void verificaNome() {FunPro = false;}

   advice "FimDeclaracao" : void verificaTipo() {}

   advice "FimDeclaracao" : void geraCodigo() {}

   advice "FimDeclaracao" : void verificaFunPro() {}

   advice "FimDeclaracao" : bool PossuiFunPro() {
        return false;
    }

   advice "FimDeclaracao" : void geraCodigoFunPro() {}

};

#endif
```

DeclVarAsp.ah

```
#ifndef _DECLVARASP_AH_
#define _DECLVARASP_AH_

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect DeclVarAsp {

public:
    advice "DeclVar" : void verificaNome() {
      if(!tab->insere(new Item(id->getToken_value(), var, tp->gettipo())))
        erros->push_back(new string("Identificador " + id->getToken_value()
                                    + " declarado anteriormente.\n"));
    }
```

```
    advice "DeclVar" : void verificaFunPro() {}


    advice "DeclVar" : void verificaTipo() {
        exp_inicial->verificaTipo();
        if(exp_inicial->gettipo() != tp->gettipo())
          erros->push_back(new string("Inicialização da variável "
          + id->getToken_value() + " possui valor inconsistente.\n"));
    }

    advice "DeclVar" : void geraCodigoFunPro() {}

    advice "DeclVar" : void geraCodigo() {
      if(tp->gettipo() == integer)
        (*saida) <<"  int ";
      else
        (*saida) << "  bool ";

      (*saida) <<id->getToken_value() <<" = ";
      exp_inicial->geraCodigo();
      (*saida) <<";\n";

    }

};

#endif
```

DeclConstAsp.ah

```
#ifndef _DECLCONSTASP_AH_
#define _DECLCONSTASP_AH_

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect DeclConstAsp {

public:
    advice "DeclConst" : void verificaNome() {
      if(!tab->insere(new Item(id->getToken_value(),
                      constante, tp->gettipo())))
        erros->push_back(new string("Identificador " + id->getToken_value()
                                    + " declarado anteriormente.\n"));
    }

    advice "DeclConst" : void verificaFunPro() {}


    advice "DeclConst" : void verificaTipo() {
        exp_inicial->verificaTipo();
        if(exp_inicial->gettipo() != tp->gettipo())
          erros->push_back(new string("Inicialização da constante " +
```

```
            id->getToken_value() + " possui valor inconsistente.\n"));
    }

    advice "DeclConst" : void geraCodigoFunPro() {}

    advice "DeclConst" : void geraCodigo() {
      if(tp->gettipo() == integer)
        (*saida) <<"  const int ";
      else
        (*saida) << "  const bool ";

      (*saida) <<id->getToken_value() <<" = ";
      exp_inicial->geraCodigo();
      (*saida) <<";\n";

    }
};

#endif
```

DeclFuncAsp.ah

```
#ifndef _DECLFUNCASP_AH_
#define _DECLFUNCASP_AH_

#include "../Class/Tabela.h"
#include "../Class/ItemFuncProc.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect DeclFuncAsp {

public:
    advice "DeclFunc" : void verificaNome() {
      if(!tab->insere(new ItemFuncProc(FuncNome->getToken_value(), function,
                                       Type1->gettipo(), Type2->gettipo())))
        erros->push_back(new string("Identificador "
            + FuncNome->getToken_value() + " declarado anteriormente.\n"));
    }

    advice "DeclFunc" : void verificaFunPro() {
        Tabela *temp = tab;
        tab = new Tabela();
        tab->insere(new ItemFuncProc(FuncNome->getToken_value(),
                    function, Type1->gettipo(), Type2->gettipo()));
        if(!tab->insere(new Item(ParNome->getToken_value(), var,
                        Type2->gettipo())))
          erros->push_back(new string("O parâmetro "
         + ParNome->getToken_value() + " possui o mesmo nome da função.\n"));
        else {
                Func_Decl->verificaNome();
                FuncExp->verificaNome();
        }
```

```
        if(Func_Decl->PossuiFunPro())
          erros->push_back(new string("Dentro de funções ou procedimentos" +
          "não podem ser declarados outras funções ou procedimentos.\n"));

        if(erros->size() == 0) {
            Func_Decl->verificaTipo();
                FuncExp->verificaTipo();
        }

        if(FuncExp->gettipo() != Type1->gettipo())
          erros->push_back(new string("Tipo de retorno da função" +
                          " é inválido.\n"));

        delete tab;

        tab = temp;
    }


    advice "DeclFunc" : void verificaTipo() {}

    advice "DeclFunc" : void geraCodigoFunPro() {
        if(Type2->gettipo() == integer)
          (*saida) <<"int " <<FuncNome->getToken_value() <<"(";
        else
          (*saida) <<"bool " <<FuncNome->getToken_value() <<"(";

        if(Type1->gettipo() == integer)
          (*saida) <<"int " <<ParNome->getToken_value() <<") {\n";
        else
          (*saida) <<"bool " <<ParNome->getToken_value() <<") {\n";

        Func_Decl->geraCodigo();
        (*saida) <<"  return ";
        FuncExp->geraCodigo();
        (*saida) <<";\n";
        (*saida) <<"}\n\n";
    }

    advice "DeclFunc" : void geraCodigo() {}

};

#endif
```

DeclProcAsp.ah

```
#ifndef _DECLPROCASP_AH_
#define _DECLPROCASP_AH_

#include "../Class/Tabela.h"
#include "../Class/ItemFuncProc.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;
```

```
aspect DeclProcAsp {

public:
    advice "DeclProc" : void verificaNome() {
      if(!tab->insere(new ItemFuncProc(ProcNome->getToken_value(), procedure,
         nothing, tp->gettipo())))
        erros->push_back(new string("Identificador "
        + ProcNome->getToken_value() + " declarado anteriormente.\n"));
    }

    advice "DeclProc" : void verificaFunPro() {
        Tabela *temp = tab;
        tab = new Tabela();
        tab->insere(new ItemFuncProc(ProcNome->getToken_value(), procedure,
                  nothing, tp->gettipo()));
        if(!tab->insere(new Item(ParNome->getToken_value(), var,
         tp->gettipo())))
          erros->push_back(new string("O parâmetro "
        + ParNome->getToken_value() + " possui o mesmo nome da função.\n"));
        else {
                Proc_Decl->verificaNome();
                ProcCom->verificaNome();
        }

        if(Proc_Decl->PossuiFunPro())
          erros->push_back(new string("Dentro de funções ou procedimentos"
        + " não podem ser declarados outras funções ou procedimentos.\n"));

        if(erros->size() == 0) {
            Proc_Decl->verificaTipo();
                ProcCom->verificaTipo();
        }

        delete tab;

        tab = temp;

    }

    advice "DeclProc" : void verificaTipo() {}

    advice "DeclProc" : void geraCodigoFunPro() {

        (*saida) <<"void " <<ProcNome->getToken_value() <<"(";

        if(tp->gettipo() == integer)
          (*saida) <<"int " <<ParNome->getToken_value() <<") {\n";
        else
          (*saida) <<"bool " <<ParNome->getToken_value() <<") {\n";

        Proc_Decl->geraCodigo();
        ProcCom->geraCodigo();
        (*saida) <<"}\n\n";
    }

    advice "DeclProc" : void geraCodigo() {}

};

#endif
```

TypeAst.ah

```
#ifndef _TYPEASP_AH_
#define _TYPEASP_AH_

#include "../Class/Item.h"

aspect TypeAsp {

public:
    advice "Type" : virtual int gettipo() = 0;

    advice "Inteiro" : int gettipo() {
        return integer;
    }

    advice "Booleano" : int gettipo() {
      return boolean;
    }

    advice "Inteiro" || "Booleano" : void verificaNome() {}

    advice "Inteiro" || "Booleano" : void verificaFunPro() {}

    advice "Inteiro" || "Booleano" : void verificaTipo() {}

    advice "Inteiro" || "Booleano" : void geraCodigoFunPro() {}

    advice "Inteiro" || "Booleano" : void geraCodigo() {}

};

#endif
```

AtrAsp.ah

```
#ifndef _ATRASP_AH_
#define _ATRASP_AH_

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect AtrAsp {

public:
    advice "Atr" : void verificaNome() {
      Item *it = tab->busca(id->getToken_value());
      if(it != NULL) {
        if(it->gettipo_construcao() == var)
          Lado_dir->verificaNome();
        else
          erros->push_back(new string("Identificador "
           + id->getToken_value() + " não é uma variável.\n"));
```

```
    }
    else
      erros->push_back(new string("Identificador "
      + id->getToken_value() + " não foi declarado anteriormente.\n"));

  }

  advice "Atr" : void verificaFunPro() {}


  advice "Atr" : void verificaTipo() {
    Lado_dir->verificaTipo();
    Item *it = tab->busca(id->getToken_value());
    if(it->gettipo_construcao() != var)
      erros->push_back(new string(id->getToken_value()
                       + " não é uma variável.\n"));
    else
      if(it->gettipo_valor() != Lado_dir->gettipo())
        erros->push_back(new string("Atribuição de "
        + id->getToken_value() + " possui tipo inconsistente.\n"));

  }

  advice "Atr" : void geraCodigoFunPro() {}

  advice "Atr" : void geraCodigo() {
    (*saida) <<"  " << id->getToken_value() <<" = ";
    Lado_dir->geraCodigo();
    (*saida) <<";\n";
  }

};

#endif
```

IfComandoAsp.ah

```
#ifndef _IFCOMANDOASP_AH_
#define _IFCOMANDOASP_AH_

#include "../Class/Item.h"
#include <fstream>

using namespace std;

extern ofstream *saida;

aspect IfComandoAsp {

public:
  advice "IfComando" : void verificaNome() {
    Condicao->verificaNome();
    Com_Then->verificaNome();
    Com_Else->verificaNome();
  }

  advice "IfComando" : void verificaFunPro() {}

  advice "IfComando" : void verificaTipo() {
    Condicao->verificaTipo();
    if(Condicao->gettipo() == boolean) {
```

```
        Com_Then->verificaTipo();
        Com_Else->verificaTipo();
      }
      else
        erros->push_back(new string("A condição do comando if "
                                    + " não é um booleano.\n"));
    }

    advice "IfComando" : void geraCodigoFunPro() {}

    advice "IfComando" : void geraCodigo() {
      (*saida) <<"  if(";
      Condicao->geraCodigo();
      (*saida) <<"){\n";
      Com_Then->geraCodigo();
      (*saida) <<"  }\n";
      (*saida) <<"  else{\n";
      Com_Else->geraCodigo();
      (*saida) <<"  }\n";
    }

};

#endif
```

WhileComandoAsp.ah

```
#ifndef _WHILECOMANDOASP_AH_
#define _WHILECOMANDOASP_AH_

#include <fstream>
#include "../Class/Item.h"

using namespace std;

extern ofstream *saida;

aspect WhileComandoAsp {

public:
    advice "WhileComando" : void verificaNome() {
      Condicao->verificaNome();
      Com_While->verificaNome();
    }

    advice "WhileComando" : void verificaFunPro() {}

    advice "WhileComando" : void verificaTipo() {
      Condicao->verificaTipo();
      if(Condicao->gettipo() == boolean)
        Com_While->verificaTipo();
      else
        erros->push_back(new string("A condição do comando while não"
                         + " é um booleano.\n"));
    }

    advice "WhileComando" : void geraCodigoFunPro() {}

    advice "WhileComando" : void geraCodigo() {
      (*saida) <<"  while(";
      Condicao->geraCodigo();
```

```
      (*saida) <<"){\n";
      Com_While->geraCodigo();
      (*saida) <<"  }\n";
    }

};

#endif
```

OutputComandoAsp.ah

```
#ifndef _OUTPUTCOMANDOASP_AH_
#define _OUTPUTCOMANDOASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect OutputComandoAsp {

public:
    advice "OutputComando" : void verificaNome() {
      saidaExp->verificaNome();
    }

    advice "OutputComando" : void verificaFunPro() {}

    advice "OutputComando" : void verificaTipo() {
      saidaExp->verificaTipo();
    }

    advice "OutputComando" : void geraCodigoFunPro() {}

    advice "OutputComando" : void geraCodigo() {
      (*saida) <<"  cout <<";
      saidaExp->geraCodigo();
      (*saida) <<"<<endl;\n";
    }

};

#endif
```

BlocoAsp.ah

```
#ifndef _BLOCOASP_AH_
#define _BLOCOASP_AH_


aspect BlocoAsp {

public:
    advice "Bloco" : void verificaNome() {
      lista_comandos->verificaNome();
    }

    advice "Bloco" : void verificaFunPro() {}
```

```
    advice "Bloco" : void verificaTipo() {
      lista_comandos->verificaTipo();
    }

    advice "Bloco" : void geraCodigoFunPro() {}


    advice "Bloco" : void geraCodigo() {
      lista_comandos->geraCodigo();
    }

};

#endif
```

ChamaProcAsp.ah

```
#ifndef _CHAMAPROCASP_AH_
#define _CHAMAPROCASP_AH_

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;


aspect ChamaProcAsp {

public:
    advice "ChamaProc" : void verificaNome() {
      Item *it = tab->busca(ProcNome->getToken_value());
      if(it != NULL) {
        if(it->gettipo_construcao() == procedure)
          Parametro->verificaNome();
        else
          erros->push_back(new string("Identificador "
           + ProcNome->getToken_value() + " não é uma procedure.\n"));
      }
      else
        erros->push_back(new string("Identificador "
         + ProcNome->getToken_value() + " não foi declarado anteriormente.\n"));
    }

    advice "ChamaProc" : void verificaFunPro() {}

    advice "ChamaProc" : void verificaTipo() {
      ItemFuncProc *ifp = (ItemFuncProc *) tab->busca(ProcNome->getToken_value());
      Parametro->verificaTipo();
      if(ifp->getTipoParametro() != Parametro->gettipo())
        erros->push_back(new string("Passagem de parâmetro do procedimento "
         + ProcNome->getToken_value() + " não é válida.\n"));

    }

    advice "ChamaProc" : void geraCodigoFunPro() {}
```

```
    advice "ChamaProc" : void geraCodigo() {
      (*saida) <<"  " << ProcNome->getToken_value() <<"(";
      Parametro->geraCodigo();
      (*saida) <<");\n";
    }

};

#endif
```

ExpressoesAsp.ah

```
#ifndef _EXPRESSOESASP_AH_
#define _EXPRESSOESASP_AH_


#include "../Class/Item.h"
#include <string>
#include <vector>

using namespace std;

extern vector<string *> *erros;

aspect ExpressoesAsp {

private:
   advice "Numero" || "ConstanteTrue" || "ConstanteFalse" || "Adicao"
          || "Subtracao" || "IdExp" || "Multiplicacao" || "MenorExp"
          || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
          || "IfExpressao" || "ChamaFunc" || "Divisao"
          || "AndExp" : int tipo;

public:

    advice "Numero" || "ConstanteTrue" || "ConstanteFalse"
           || "ReadValor" : void verificaNome() {}

    advice "ConstanteTrue" || "ConstanteFalse" : void verificaTipo() {
        tipo = boolean;
    }

    advice "Adicao" || "Subtracao" || "IgualExp"
        || "Multiplicacao" || "MenorExp" || "Divisao"
        || "AndExp"; : void verificaNome() {
        Exp1->verificaNome();
        Exp2->verificaNome();
    }

    advice "Adicao" || "Subtracao" || "Multiplicacao"
       || "Divisao" : void verificaTipo() {
        Exp1->verificaTipo();
        Exp2->verificaTipo();
        if(Exp1->gettipo() != integer || Exp2->gettipo() != integer)
          erros->push_back(new string("Operacao possui tipos inconsistente.\n"));

        tipo = integer;
    }
```

```
    advice "IgualExp" || "MenorExp" || "AndExp" : void verificaTipo() {
        Exp1->verificaTipo();
        Exp2->verificaTipo();
        if(Exp1->gettipo() !=  Exp2->gettipo())
          erros->push_back(new string("Igualdade possui tipos inconsistente.\n"));

        tipo = boolean;
    }


    advice "Expressoes" : virtual int gettipo() = 0;

    advice "Expressoes" : virtual void settipo(int ) = 0;

    advice "Numero" || "ConstanteTrue" || "ConstanteFalse" || "Adicao"
          || "Subtracao" || "IdExp" || "Multiplicacao" || "MenorExp"
          || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
          || "IfExpressao" || "ChamaFunc" || "Divisao"
          || "AndExp" : int gettipo() {
                return tipo;
    }

    advice "Numero" || "ConstanteTrue" || "ConstanteFalse" || "Adicao"
          || "Subtracao" || "IdExp" || "Multiplicacao" || "MenorExp"
          || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
          || "IfExpressao" || "ChamaFunc" || "Divisao"
          || "AndExp" : void settipo(int t) {
                tipo = t;
    }


    advice "Numero" || "ConstanteTrue" || "ConstanteFalse" || "Adicao"
          || "Subtracao" || "IdExp" || "Multiplicacao" || "MenorExp"
          || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
          || "IfExpressao" || "ChamaFunc" || "Divisao"
          || "AndExp" : void verificaFunPro() {}


    advice "Numero" || "ConstanteTrue" || "ConstanteFalse" || "Adicao"
          || "Subtracao" || "IdExp" || "Multiplicacao" || "MenorExp"
          || "Negacao" || "IgualExp" || "ParExp" || "ReadValor"
          || "IfExpressao" || "ChamaFunc" || "Divisao"
          || "AndExp" : void geraCodigoFunPro() {}


};

#endif
```

NumeroAsp.ah

```
#ifndef _NUMEROASP_AH_
#define _NUMEROASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect NumeroAsp {
```

```
public:

  advice "Numero" : void verificaTipo() {
        tipo = integer;
  }

  advice "Numero" : void geraCodigo() {
    (*saida) <<ConstanteInt0->getToken_value();
  }

};

#endif
```

ConstanteTrueAst.ah

```
#ifndef _CONSTANTETRUEASP_AH_
#define _CONSTANTETRUEASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect ConstanteTrueAsp {

public:

  advice "ConstanteTrue" : void geraCodigo() {
    (*saida) <<True0->getToken_value();
  }

};

#endif
```

ConstanteFalseAsp.ah

```
#ifndef _CONSTANTEFALSEASP_AH_
#define _CONSTANTEFALSEASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect ConstanteFalseAsp {

public:

  advice "ConstanteFalse" : void geraCodigo() {
    (*saida) <<False0->getToken_value();
  }

};

#endif
```

AdicaoAsp.ah

```
#ifndef _ADICAOASP_AH_
#define _ADICAOASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect AdicaoAsp {

public:

  advice "Adicao" : void geraCodigo() {
        Exp1->geraCodigo();
    (*saida) <<" + ";
    Exp2->geraCodigo();
  }

};

#endif
```

SubtracaoAsp.ah

```
#ifndef _SUBTRACAOASP_AH_
#define _SUBTRACAOASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect SubtracaoAsp {

public:

  advice "Subtracao" : void geraCodigo() {
    Exp1->geraCodigo();
    (*saida) <<" - ";
    Exp2->geraCodigo();
  }

};

#endif
```

MultiplicacaoAsp.ah

```
#ifndef _MULTIPLICACAOASP_AH_
#define _MULTIPLICACAOASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;
```

```
aspect MultiplicacaoAsp {

public:
     advice "Multiplicacao" : void geraCodigo() {
        Exp1->geraCodigo();
        (*saida) <<" * ";
        Exp2->geraCodigo();
     }

};

#endif
```

DivisaoAsp.ah

```
#ifndef _DIVISAOASP_AH_
#define _DIVISAOASP_AH_


#include <fstream>

using namespace std;

extern ofstream *saida;

aspect DivisaoAsp {

public:
     advice "Divisao" : void geraCodigo() {
        Exp1->geraCodigo();
        (*saida) <<" / ";
        Exp2->geraCodigo();
     }

};

#endif
```

MenorExpAsp.ah

```
#ifndef _MENOREXPASP_AH_
#define _MENOREXPASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect MenorExpAsp {

public:
     advice "MenorExp" : void geraCodigo() {
        Exp1->geraCodigo();
        (*saida) <<" < ";
        Exp2->geraCodigo();
     }

};
```

```
#endif
```

IgualExpAsp.ah

```
#ifndef _IGUALEXPASP_AH_
#define _IGUALEXPASP_AH_

#include <fstream>

using namespace std;

extern ofstream *saida;

aspect IgualExpAsp {

public:
     advice "IgualExp" : void geraCodigo() {
        Exp1->geraCodigo();
        (*saida) <<" == ";
        Exp2->geraCodigo();
     }

};

#endif
```

AndExpAsp.ah

```
#ifndef _ANDEXPASP_AH_
#define _ANDEXPASP_AH_


#include <fstream>

using namespace std;

extern ofstream *saida;

aspect AndExpAsp {

public:
     advice "AndExp" : void geraCodigo() {
        Exp1->geraCodigo();
        (*saida) <<" && ";
        Exp2->geraCodigo();
     }

};

#endif
```

NegacaoAsp.ah

```
#ifndef _NEGACAOASP_AH_
#define _NEGACAOASP_AH_

#include "../Class/Item.h"
#include <fstream>
```

```
#include <string>
#include <vector>

using namespace std;

extern vector<string *> *erros;
extern ofstream *saida;

aspect NegacaoAsp {

public:
    advice "Negacao" : void verificaNome() {
      Exp0->verificaNome();
    }

    advice "Negacao" : void verificaTipo() {
      if(Exp0->gettipo() != boolean)
        erros->push_back(new string("Expressao em Not não é um boolean.\n"));

      tipo = boolean;
    }

     advice "Negacao" : void geraCodigo() {
       (*saida) <<"!";
       Exp0->geraCodigo();
     }

};

#endif
```

IdExpAsp.ah

```
#ifndef _IDEXPASP_AH_
#define _IDEXPASP_AH_

#include "../Class/Tabela.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
extern ofstream *saida;

aspect IdExpAsp {

public:
    advice "IdExp" : void verificaNome() {
      Item *it = tab->busca(id->getToken_value());
      if(it == NULL)
        erros->push_back(new string("Identificador "
         + id->getToken_value() + " não foi declarado anteriormente.\n"));

    }

    advice "IdExp" : void verificaTipo() {
      Item *it = tab->busca(id->getToken_value());
      tipo = it->gettipo_valor();
```

```
    }

     advice "IdExp" : void geraCodigo() {
       (*saida) <<id->getToken_value();
     }

};

#endif
```

ParExpAsp.ah

```
#ifndef _PAREXPASP_AH_
#define _PAREXPASP_AH_


#include <fstream>

using namespace std;


extern ofstream *saida;

aspect ParExpAsp {

public:
    advice "ParExp" : void verificaNome() {
      Exp0->verificaNome();
    }

    advice "ParExp" : void verificaTipo() {
      Exp0->verificaTipo();
      tipo = Exp0->gettipo();
    }

     advice "ParExp" : void geraCodigo() {
       (*saida) <<"(";
       Exp0->geraCodigo();
       (*saida) <<")";
     }

};

#endif
```

ChamaFuncAsp.ah

```
#ifndef _CHAMAFUNCASP_AH_
#define _CHAMAFUNCASP_AH_

#include "../Class/Tabela.h"
#include "../Class/ItemFuncProc.h"
#include <string>
#include <vector>
#include <fstream>

using namespace std;

extern Tabela *tab;
extern vector<string *> *erros;
```

```
extern ofstream *saida;


aspect ChamaFuncAsp {

public:
    advice "ChamaFunc" : void verificaNome() {
      Item *it = tab->busca(FuncNome->getToken_value());
      if(it != NULL) {
        if(it->gettipo_construcao() == function)
          Parametro->verificaNome();
        else
          erros->push_back(new string("Identificador " + FuncNome->getToken_value()
                          + " não é uma função.\n"));
      }
      else
        erros->push_back(new string("Identificador " + FuncNome->getToken_value()
         + " não foi declarado anteriormente.\n"));
    }


    advice "ChamaFunc" : void verificaTipo() {
      ItemFuncProc *ifp = (ItemFuncProc *) tab->busca(FuncNome->getToken_value());
      Parametro->verificaTipo();
      if(ifp->getTipoParametro() != Parametro->gettipo())
        erros->push_back(new string("Passagem de parâmetro do procedimento "
         + FuncNome->getToken_value() + " não é válida.\n"));

      tipo = ifp->gettipo_valor();

    }


    advice "ChamaFunc" : void geraCodigo() {
      (*saida) <<"  " << FuncNome->getToken_value() <<"(";
      Parametro->geraCodigo();
      (*saida) <<")";
    }

};

#endif
```

ReadValorAsp.ah

```
#ifndef _READVALORASP_AH_
#define _READVALORASP_AH_

#include "../Class/Item.h"
#include <fstream>

using namespace std;

extern ofstream *saida;

aspect ReadValorAsp {

public:


    advice "ReadValor" : void verificaTipo() {
```

```
      tipo = integer;
    }

     advice "ReadValor" : void geraCodigo() {
       (*saida) <<"  Read()";
     }



};

#endif
```

IfExpressaoAsp.ah

```
#ifndef _IFEXPRESSAOASP_AH_
#define _IFEXPRESSAOASP_AH_

#include "../Class/Item.h"
#include <fstream>

using namespace std;

extern ofstream *saida;

aspect IfExpressaoAsp {

public:
    advice "IfExpressao" : void verificaNome() {
      Condicao->verificaNome();
      Exp_Then->verificaNome();
      Exp_Else->verificaNome();
    }

    advice "IfExpressao" : void verificaTipo() {
      Condicao->verificaTipo();
      if(Condicao->gettipo() == boolean) {
        Exp_Then->verificaTipo();
        Exp_Else->verificaTipo();
        if(Exp_Then->gettipo() == Exp_Else->gettipo())
          tipo = Exp_Then->gettipo();
        else {
          erros->push_back(new string("As expressões do if não possuem"
                           + " o mesmo tipo.\n"));
          tipo = nothing;
        }
      }
      else {
        erros->push_back(new string("A condição do comando if"
                         + " não é um booleano.\n"));
        tipo = nothing;
      }

    }

    advice "IfExpressao" : void geraCodigo() {
      Condicao->geraCodigo();
      (*saida) <<" ? ";
      Exp_Then->geraCodigo();
      (*saida) <<" : ";
      Exp_Else->geraCodigo();
```

```
    }

};

#endif
```

TokenAst.ah

```
#ifndef _TOKENASP_AH_
#define _TOKENASP_AH_


aspect TokenAsp {

        pointcut ptokens() = "Identificador" || "ConstanteInt" || "True"
                             || "False";

        advice ptokens() : void verificaNome() {}

        advice ptokens() : void verificaFunPro() {}

        advice ptokens() : void verificaTipo() {}

        advice ptokens() : void geraCodigoFunPro() {}

        advice ptokens() : void geraCodigo() {}


};

#endif
```

Referências Bibliográficas

[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] M. Anlauff, P. W. Kutter and A. Pierantonio, *Tool Suport for Language Design and Prototyping with Montages*. In Proceedings of Compiler Construction (CC'99). Springer, LNCS, 1999.

[3] M. Anlauff, *Xasm - An Extensible, Component-Based Abstract State Machines Language*. In Proceedings of the ASM 2000 Workshop, pp 1-21, Monte Veritá, Switzerland, Março 2000.

[4] AspectC++. *http://www.aspectc.org/*.

[5] Aspectj.org. *http://www.aspectj.org/*.

[6] D. Bèauquier and A. Slissenko, *On Semantics of Algorithms with Continuous Time*. Technical Report 97-15, Dept. of Informatics, Université Paris-12, October 1997.

[7] D. Bèauquier and A. Slissenko, *The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages*. In M. Bidoit and M. Dauchet, editors, TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, volume 1214 of LNCS, pages 201-212. Springer, 1997.

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

[9] E. Börger, *Why Use Evolving Algebras for Hardware and Software Engineering?*. In M. Bartosek, J. Staudek, and J. Wiederman, editors, Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and

Practice of Informatics, volume 1012 of LNCS, pages 236-271. Springer, 1995.

[10] E. Börger and U. Glässer, *Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras*. In Y. Gurevich and E. Börger, editors, Evolving Algebras - Mini-Course, BRICS Technical Report (BRICS-NS-95-4), pages 128-153. University of Aarhus, Denmark, July 1995.

[11] E. Börger, Y. Gurevich, and D. Rosenzweig, *The Bakery Algorithm: Yet Another Specification and Verification*. In E. Börger, editor, Specification and Validation Methods, pages 231-243. Oxford University Press, 1995.

[12] E. Börger and J. Huggins, *Abstract State Machines 1988-1998: Commented ASM Bibliography*. Bulletin of EATCS, 64:105-127, February 1998.

[13] E. Börger and S. Mazzanti, *A Practical Method for Rigorously Controllable Hardware Design*. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, ZUM'97: The Z Formal Specification Notation, volume 1212 of LNCS, pages 151-187. Springer, 1997.

[14] E. Börger and D. Rosenzweig, *A Mathematical Definition of Full Prolog*. In Science of Computer Programming, volume 24, pages 249-286. North-Holland, 1994.

[15] E. Börger and W. Schulte, *Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation*. In L. Brim and J. Gruska and J. Zlatuska, editor, Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98, Brno, Czech Republic, number 1450 in LNCS. Springer, August 1998.

[16] E. Börger and W. Schulte, *Programmer Friendly Modular Definition of the Semantics of Java*. In J. Alves-Foss, editor, Formal Syntax and Semantics of Java, LNCS. Springer, 1998.

[17] M. Brukman, A. C. Myers. *PPG: a parser generator for extensible grammars*, 2003. Available at www.cs.cornell.edu/Projects/polyglot/ppg.html.

[18] G. D. Castillo, I. Durdanović and U. Glässer, *An Evolving Algebra Abstract Machine*. In H. Kleine Büning, editor, Proceedings of the Annual

Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of LNCS, pages 191-214. Springer, 1996.

[19] G. D. Castillo, I. Durdanovic, and U. Glasser, *The Evolving Algebra Interpreter Version 2.0*, 191-214, 1996.

[20] G. D. Castillo, *The ASM Workbench: na Open and Extensible Tool Enviroment for Abstract State Machine*. In Proceedings of the 28th Annual Conference of the German Society of Computer Science. Technical Report 1998.

[21] G. D. Castillo, *Towards Comprehensive Tool Support for Abstract State Machines*. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, Applied Formal Methods - FM-Trends 98, volume 1641 of LNCS, pages 311-325. Springer-Verlag, 1999.

[22] Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson and J. S. Ong, *Structuring operating system aspects: using AOP to improve OS structure modularity*. In Communications of the ACM, 44 (10), 2001.

[23] Y. Coady, G. Kiczales, M. Feeley and G. Smolyn, *Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code*. 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9), pages 88-98. ACM Press, 2001.

[24] C. A. Constantinides, A. Bader, T. H. Elrad, M. E. Fayad, and P. Netinant, *Designing an Aspect-Oriented Framework in an Object-Oriented Environment*. ACM Computing Surveys Symposium on Application Frameworks, M.E. Fayad, Editor, Vol. 32, No. 1, March 2000.

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

[26] I. Durdanović, *From Operational Specifications to Real Architectures*. Draft of PhD Thesis (NEC Research Institute Princeton), March 2, 2000.

[27] Eclipse.org - *http://www.eclipse.org*.

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

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

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

[31] U. Glässer and R. Karges, *Abstract State Machine Semantics of SDL.* Journal of Universal Computer Science, 3(12):1382-1414, 1997.

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

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

[34] Y. Gurevich and J. Huggins, *The Semantics of the C Programming Language.* In E. Börger, H. Kleine BÄuning, G. Jäger, S. Martini, and M. M. Richter, editors, Computer Science Logic, volume 702 of LNCS, pages 274-309. Springer, 1993.

[35] Y. Gurevich and J. Huggins, *The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions.* In Proceedings of CSL'95 (Computer Science Logic), volume 1092 of LNCS, pages 266-290. Springer, 1996.

[36] Y. Gurevich and R. Mani, *Group Membership Protocol: Specification and Verification.* In E. Börger, editor, Specification and Validation Methods, pages 295-328. Oxford University Press, 1995.

[37] Michael J. C. Gordon, The Denotational Description of Programming Languages: An Introduction, , 1979.

[38] 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.

[39] G. Hedin, *Reference Attributed Grammars.* In D. Parigot and M. Mernik, editors, Second Workshop on Attribute Grammars and their Applications, WAGA 99, pages 153-172, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[40] 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.

[41] J. Huggins and R. Mani, *The Evolving Algebra Interpreter Version 2.0.* Documentation of the Michigan Evolving Algebra Interpreter, available electronically at ftp://ftp.eecs.umich.edu/groups/Ealgebras/interp2.tar.Z. 1995

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

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

[44] 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, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 2001.

[45] 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, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

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

[47] P.W. Kutter and A. Pierantonio, Montages specifications of realistic programming languages. In Journal of universal computer science, Vol. 3, No 5 (1997), pp 416-442, Springer.

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

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

[50] 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.

[51] R. C. Martin, The Open Closed Principle, Technical Report, Jan 1996.

[52] B. Meyer,, Object-Oriented Software Construction, Prentice Hall, 1994.

[53] Microsoft Research, *AsmL: The Abstract State Machine Language*, Foundations of Software Engineering, ©Microsoft Corporation, 2002.

[54] Microsoft Research, *Introducing AsmL: A Tutorial for the Abstract state Machine Language*. Foundations of Software Engineering, ©Microsoft Corporation, 2002.

[55] 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.

[56] F. F. de Oliveira, R. S. Bigonha, and M. A. S. Bigonha, *Otimização de Código em Ambiente de Semântica Formal Executável Baseado em ASM*. Proceedings of 8th Brazilian Symposium on Programming Languages, pages 172-185, May 2004.

[57] 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.

[58] H. Ossher and P. Tarr, *HyperJ: Multi-Dimensional Separation of Concerns for Java*. 22nd international conference on Software engineering, pages 734-737. ACM Press, 2000.

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

[60] R. R. Raje, M. Zhong and T. Wang, *Case Study: A Distributed Concurrent System with AspectJ*. ACM Applied Computing Review, Vol. 9, No. 2, pp. 17 - 23, 2001.

[61] A. Rashid and R. Chitchyan, *Persistence as an Aspect*. International Conference on Aspect-Oriented Software Development, March 2003.

[62] J. Schmid, *Introduction do AsmGofeer*. Technical Report, Siemens Corporate Technology. Munich, Março, 2001.

[63] S. Soares, E. Laureano and P. Borba, *Implementing distribution and persistence aspects with AspectJ*. ACM SIGPLAN Notices, v.37 n.11, November 2002.

[64] O. Spinczyk, A. Gal and W. Schröder-Preikschat, *AspectC++: An Aspect-Oriented Extension to the C++ Programming Language*. In Proceedings of the 40th International Conference on Tools Pacific, pages 53-60. Australian Computer Society, Inc. , 2002.

[65] SUN Microsystems, *JavaCC, the SUN Java Compiler Compiler*. http://www.suntest.com/JavaCC.

[66] SUN Microsystems, http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javac.html

[67] F. Tirelo, M. A. Maia, R. S. Bigonha, and V. O. Di Iorio. *Tutorial sobre Máquinas de Estado Abstratas*. In Anexo dos Proceedings fo the III Simpósio Brasileiro de Linguagens de Programação, pages 1-30, Porto Alegre, RS, Maio 1999. SBC.

[68] 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.

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

[70] 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.

[71] J. M. W. Visser, *Evolving algebras*. Master's thesis, Delft University of Technology, Delt, 1996

[72] C. Wallace, *The Semantics of the C++ Programming Language*. In E. Börger, editor, Specification and Validation Methods, pages 131-164. Oxford University Press, 1995.

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