ESPECIALIZAÇÃO DE LINGUAGENS ORIENTADAS A ASPECTOS BASEADA EM EXTENSIBILIDADE DE GRAMÁTICAS

LEONARDO VIEIRA DOS SANTOS REIS

ESPECIALIZAÇÃO DE LINGUAGENS ORIENTADAS A ASPECTOS BASEADA EM EXTENSIBILIDADE DE GRAMÁTICAS

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

Orientador: Roberto da Silva Bigonha
Co-Orientador: Vladimir Oliveira Di Iorio

Belo Horizonte

Março de 2010

Reis, Leonardo Vieira dos Santos
R375e Especialização de linguagens orientadas a aspectos baseada em extensibilidade de gramáticas / Leonardo Vieira dos Santos Reis. — Belo Horizonte, 2010
xxii, 110 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Roberto da Silva Bigonha

Co-Orientador: Vladimir Oliveira Di Iorio

1. Programação orientada a objetos - Teses. 2. Linguagem Extensível - Teses. 3. Linguagem AspectJ - Teses. I. Orientador II. Co-Orientador. III. Título.

CDU 519.6*33(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Especialização de linguagens orientadas a aspéctos baseada em extensibilidade de gramáticas

LEONARDO VIEIRA DOS SANTOS REIS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. VLADIMIR OLIVEIRA DI IORIO - Co-orientador
Departamento de Informática - UFV

PROF. MARCELO DE ALMEIDA MAIA
Departamento de Ciência da Computação - UFU

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 12 de março de 2010.

Para Helen e Emmanuel pelo amor e paciência.

Agradecimentos

Agradeço ao criador pela vida. Aos meus pais Ednaldo e Cristina pela educação, amor e carinho. Agradeço, também, à todos os meus familiares, minha avó Anizia, meus tios(as) Móises, Rosival, Pedro, Idma, Néia, Marlene, Argentina, Vera, meus irmãos Diego, Milton e Maísa, e primos(as), que sempre me apoiaram e contribuiram para eu ter conseguido concretizar esta etapa da minha vida, em especial meu tio Nilton que tem sido um segundo pai para mim. Ao meu tio Magnaldo e minha avó Helena, que já partiram desta vida e não me viram finalizar mais esta etapa, e, tenho certeza que estariam felizes com mais essa conquista.

Agradeço aos meus amigos de república, Henrique, Luis Felipe e Hadriel, pela boa convivência. Aos amigos de mestrado e laboratório, Pedro, Sérgio, Elton, André, Andrei, Fernando, Rodrigo, César, Terra, Gabriel e Fillipe pelo excelente ambiente de trabalho e estudo. Aos amigos do grupo Pilares de Luz pelas boas energias.

Agradeço aos meus orientadores Roberto Bigonha, Vladimir Di Iorio e Mariza Bigonha pelo inúmeros conselhos e revisões. Obrigado Vladimir por ter sido, além de professor e orientador, um amigo e conselheiro.

Obrigado Helen e Emmanuel, que sempre estiveram ao meu lado me apoiando e dando forças, pelo amor e pela felicidade que tenho de fazer parte de suas vidas.

Resumo

O desenvolvimento dos recursos computacionais e o surgimento da indústria de software trouxeram a necessidade de produzir softwares rapidamente e mais complexos. Com o aumento da complexidade dos softwares, surgiram problemas de manutenção, pois a complexidade e o tamanho dos programas aumentaram, o que dificulta encontrar e corrigir erros. Neste contexto, surgiu a necessidade de desenvolver ferramentas que aumentem a produtividade e facilitem a manutenção dos softwares. Programação Orientada a Aspectos e Linguagens de Domínio Específico são metodologias de programação que visam aumentar a produtividade e facilitar a manutenção de código. No entanto criar extensões e embarcar linguagens de domínio específico orientadas a aspectos (união das duas técnicas citadas) em linguagens de propósito geral, e.g., AspectJ, AspectC, etc., não são tarefas simples.

Este trabalho apresenta a linguagem XAJ, a primeira linguagem extensível orientada a aspectos que se tem conhecimento, a qual permite estender sua sintaxe concreta, possibilitando criar extensões e linguagens de domínio específico orientadas a aspectos embarcadas. XAJ tem uma nova unidade sintática, as classes sintáticas, que encapsulam as definições das extensões, aumentando a modularidade das mesmas. Novos mecanismos de importação são definidos para o uso das extensões, permitindo que sejam usadas em escopo local, não modificando globalmente a gramática da linguagem base. As extensões definidas na linguagem XAJ são portáveis, pois não são dependentes de ferramentas de implementação, sendo definidas na própria linguagem.

Palavras Chaves: Extensões de linguagens, programação orientada a aspectos, linguagens de domínio específico.

Abstract

The development of computational resources and the rise of software industry brought the need to produce software more quickly and with higher complexity. With the increase on program size and complexity, severe problems of maintenance arise, because errors are more difficult to find and correct. So, it is important to create tools that make maintenance easier and improve the productivity of programmers. Domain-Specific Languages and Aspect-Oriented Programming can be combined to achieve these goals, building domain-specific aspect-oriented languages as extensions of languages such as AspectJ or AspectC. But building such extensions and embedding them on general-purpose aspect languages is not an easy task.

This work presents XAJ, an aspect-oriented extensible language based on AspectJ. XAJ provides features to build extensions and embedded domain-specific aspect-oriented languages. XAJ has a new compilation unit called syntax class, which encapsulates the whole definition of a language extension, improving modularity and portability of extensions. Mechanisms for symbol import allow the definition of new grammar rules with local scope. Extensions in XAJ are portable because they are completely defined with the language itself, without the need of additional tools for the implementation.

Keywords: aspect-oriented programming, domain-specific languages, languages extensions

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Linguagens de programação são os mecanismos primários para que os softwares sejam criados e para expressar um problema deve-se fragmentá-lo em termos de construções oferecida por uma linguagem. Linguagens de próposito geral, e.g., Java, C, C++, são linguagens projetadas para serem capazes de resolver qualquer problema computável, tendo construções genéricas. Porém, expressar certos domínios de aplicações em linguagens de próposito geral requer uma implementação trabalhosa, diminuindo a produtividade e deixando mais difícil a manutenção do código. Isto ocorre porque as linguagens de próposito geral não apresentam construções adequadas para expressar tais domínios.

Linguagens de Domínio Específico (*DSL*, do inglês *Domain Specific Languages*) expressam o conhecimento específico de um domínio da aplicação, sendo melhores que as linguagens de próposito geral para o domínio em questão [Deursen et al., 2000; Bravenboer et al., 2006a]. DSLs têm a sintaxe bem próxima da notação do domínio da aplicação, o que facilita a interação de especialistas do domínio com os programadores, deixa o código mais fácil de entender e de manter. Alguns benefícios do uso de DSLs: maior expressividade, facilidade para utilização, aumento da produtividade e redução nos custos de manutenção [Mernik et al., 2005; van Deursen & Klint, 1997; Kieburtz et al., 1996].

Programação Orientada a Aspectos (*AOP*, do inglês *Aspect Oriented Programming*) é um novo paradigma de programação, introduzido em 1997 por Gregor Kiczales [Kiczales et al., 1997]. A principal motivação para a criação desse paradigma é apresentar uma solução adequada para a implementação de interesses transversais. Interesses transversais são funcionalidades que não podem ser adequadamente encapsuladas em classes e ficam espalhadas por todo o sistema.

Linguagens Orientadas a Aspectos oferecem ao programador uma poderosa ferra-

menta de transformação de código, porém para algumas aplicações, os programadores necessitam somente de um subconjunto de funcionalidades das linguagens orientadas a aspectos. Além disso, em alguns casos, o poder das linguagens orientadas a aspectos pode ser considerado excessivo, permitindo que o programador viole alguns princípios básicos da programação orientada a objetos, como quebrar o encapsulamento das classes e acessar membros privados. Linguagens de domínio específico orientadas a aspectos, (*DSAL*, do inglês *Domain-Specific Aspect Languages*), são DSLs orientadas a aspectos. DSALs podem ser uma solução para diversos problemas envolvendo interesses tranversais, oferecendo poder suficiente para aplicações específicas juntamente com uma sintaxe intuitiva e produtiva.

Linguagens de domínio específico embarcadas em linguagens de propósito geral apresentam diversas vantagens [Batory et al., 1998]. No entanto, as linguagens orientadas a aspectos de propósito geral, e.g., AspectJ e AspectC, não oferecem recursos para extensão da sua própria sintaxe concreta, o que dificulta embarcar DSALs nessas linguagens. O projeto e implementação de linguagens orientadas a aspectos é um campo de atuação que está em constante mudança, com novas características sendo desenvolvidas e propostas, portanto linguagens extensíveis orientadas a aspectos são uma boa alternativa para testar novas construções para tais linguagens, além de permitir embarcar linguagens de domínios específicos orientadas a aspectos mais facilmente em linguagens de propósito geral.

Neste trabalho, é apresentada a linguagem XAJ (*eXtensible AspectJ*), uma nova linguagem que permite estender a sintaxe concreta da linguagem AspectJ, possibilitando que novas construções propostas para a linguagem AspectJ e DSALs embarcadas sejam definidas. Uma importante vantagem de XAJ em relação a outras abordagens para estender a linguagem AspectJ é o aumento da portabilidade e da modularidade nas definições das extensões.

1.1 Motivação

O desenvolvimento dos recursos computacionais e o surgimento da indústria de software trouxeram a necessidade de produzir softwares rapidamente e mais complexos. Com o aumento da complexidade dos softwares, surgiram problemas de manutenção, pois a complexidade e o tamanho dos programas aumentaram, o que dificulta encontrar e corrigir erros. Neste contexto, surgiu a necessidade de desenvolver ferramentas que aumentem a produtividade e facilitem a manutenção dos softwares.

Os maiores ganhos em produtividade no desenvolvimento de software, historica-

mente, foram alcançados por meio da criação de linguagens de alto nível e de compiladores para essas linguagens. O uso de uma linguagem de alto nível, quando comparada à linguagem de montagem (assembler), permite que os programas escritos sejam menores, simples, fáceis de entender e modificar.

O conceito de Programação Orientada a Objetos contribui para o ganho de produtividade no desenvolvimento de software, com a introdução do conceito de classe, facilitando a divisão do programa em módulos, aumentando as possibilidades de reúso de código e facilidades de manutenção. No entanto, existem certas características em um programa que não podem ser encapsuladas em uma unidade, ou módulo, e ficam espalhadas por todo o código, denominadas interesses transversais. Exemplos importantes de interesses com essas características são o registro de operações, persistência de dados e controle de transações.

A Programação Orientada a Aspectos foi criada para ser uma solução adequada para implementar interesses transversais. No entanto, podem ser necessários mais recursos que os oferecidos pelas linguagens orientadas a aspectos, e.g., AspectJ, para implementar adequadamente alguns interesses transversais, sendo necessário adicionar novas construções à linguagem para implementar tais recursos.

Um dos problemas associado à implementação de padrões de projeto usando programação orientada a objetos é o entrelaçamento do código do padrão com o código do sistema. A abordagem orientada a aspectos apresenta soluções para este problema [Hannemann & Kiczales, 2002; Lorenz, 1998]. Entretanto, implementações orientadas a aspectos de padrões como Visitor podem apresentar uma repetição excessiva de código. A seguir é apresentada uma extensão para a linguagem AspectJ que resolve de forma elegante a repetição de código em uma implementação do padrão Visitor usando a abordagem orientada a aspectos. Este exemplo evidencia as vantagens de extensões de linguagens discutidas anteriormente (produtividade e manutenção de softwares) e será usado ao longo do texto.

Visitor[Gamma et al., 1995] é um padrão de projeto usado para representar uma operação a ser executada nos elementos de uma estrutura de objetos. O Visitor permite que se crie um nova operação sem que se altere o código da classe dos elementos sobre as quais ela opera, sendo uma maneira de separar um algoritmo da estrutura de um objeto. Desse modo, novas funcionalidades podem ser adicionadas à estrutura de objetos sem a necessidade de modificá-los.

A Figura 1.1 mostra a hierarquia de classes para o padrão Visitor. Em uma implementação deste padrão, todos os elementos da estrutura devem implementar a interface *ElementVisitor* que declara o método *accept(Visitor)*. Com esse método, a classe concreta invoca a operação correta a ser executada para o elemento correspon-

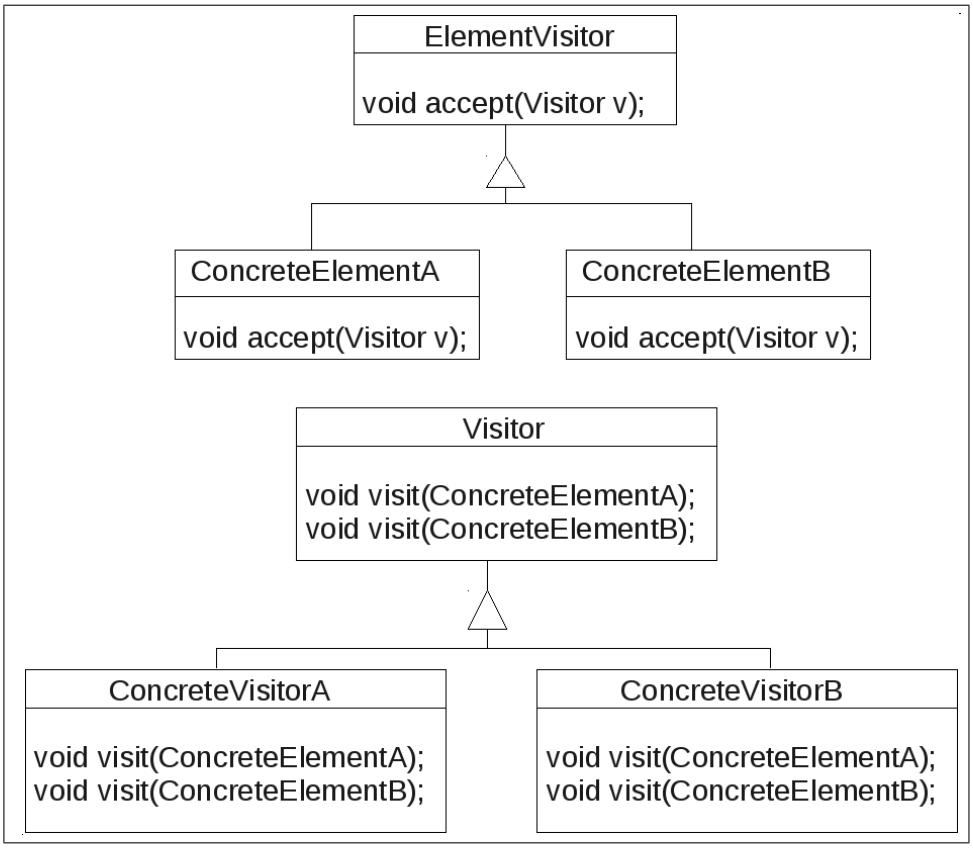

Figura 1.1. Padrão Visitor.

dente. Toda a operação a ser executada sobre a estrutura fica encapsulada em uma implementação concreta da interface *Visitor*.

Um dos problemas associados com a implementação do padrão Visitor é o entrelaçamento do código do padrão com o código do sistema. Este problema pode ser facilmente resolvido utilizando programação orientada a aspectos.

Considere que se deseja adicionar algumas operações na estrutura de classes que representa operações aritméticas, mostrada na Figura 1.2. A abordagem orientada a aspectos para implementar o padrão Visitor será usada para esta finalidade.

A Figura 1.3 mostra uma possível implementação usando a linguagem AspectJ. O código referente ao padrão fica todo encapsulado dentro do aspecto, no entanto é necessário inserir o método *accept* em todas as subclasses de *Expr*[1], pois o identificador *this* é diferente em cada classe, impossibilitando que o método fique inserido na classe base.

O código do método é igual para toda subclasse, no entanto a linguagem AspectJ não tem uma construção que interprete o identificador *this* de forma diferente em cada

[1]A inserção do método é feito usando a declaração intertipo, como é mostrada na Figura 1.3.

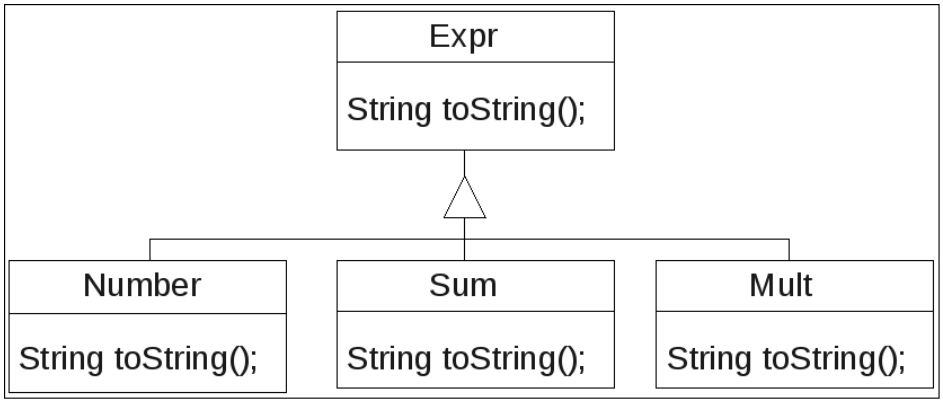

Figura 1.2. Hierarquia de Classes Para Expressões Aritméticas.

```
public aspect VisitorAspect {
  declare parents:  Expr extends ElementVisitor;
  void Sum.accept(Visitor v) {
    v.visit(this);
  }
  void Mult.accept(Visitor v) {
    v.visit(this);
  }
  void Number.accept(Visitor v) {
    v.visit(this);
  }
}
```

Figura 1.3. Versão Orientada a Aspectos do Padrão Visitor.

função e insira esse método em cada subclasse da hierarquia corretamente, assim esta repetição de código não pode ser evitada.

Chiba & Nakagawa [2004] sugerem uma nova construção para a linguagem AspectJ que poderia evitar ou minimizar a repetição de código discutida acima. Se as classes a serem visitadas são subclasses de uma classe denominada *Base*, a sintaxe da construção proposta poderia ser:

```
void Base+.accept(Visitor v) { v.visit(this); }
```

O uso de *Base+* na declaração significa que a declaração intertipo será inserida em todas as subclasses de *Base*. Este novo comando elimina a repetição de código e ainda tem uma importante vantagem: se a hierarquia de classes for alterada, incluindo ou excluindo novas classes, o código não necessita ser modificado. A extensão proposta nesta seção será referenciada nas próximas seções como *MultiIntro*, significando que ela produz múltiplas declarações intertipo. A Figura 1.4 apresenta a versão do padrão

```
public aspect VisitorAspect {
  declare parents:  Expr extends ElementVisitor;
  void Expr+.accept(Visitor v) {
    v.visit(this);
}
```

Figura 1.4. Versão Orientada a Aspectos Usando Multi-intro.

visitor usando o comando *MultiIntro*.

A linguagem XAJ, proposta neste trabalho, oferece recursos para embutir a extensão discutida nesta seção à linguagem de maneira modular. Esta extensão será usada como exemplo no Capítulo 3 para introduzir os conceitos da linguagem XAJ.

Extensões de domínio específico têm os mesmos benefícios que linguagens de domínio específico[Batory et al., 1998]. Além disso, um conjunto de extensões pode definir uma DSAL, de modo que DSALs embarcadas podem ser implementadas via extensões para a linguagem hospedeira. Assim, ao longo do texto os termos extensões para linguagens e linguagens de domínios específico embutidas serão usados indistintamente.

Nos últimos anos, o estudo e desenvolvimento de linguagens de domínio específico é um tópico que está em evidência e tem sido usado na indústria para facilitar a implementação e a manutenção de softwares, tendo Martin Fowler [Fowler, 2005] como um de seus principais defensores. A metodologia de desenvolvimento de software baseada na criação de linguagens de domínio específico embarcadas é chamada de *Programação Orientada a Linguagens* [Ward, 1994]. O desenvolvimento de ferramentas que sirvam de suporte para esta metodologia é desafiante e pesquisas estão sendo feitas neste sentido [Dmitriev, 2004; Fowler, 2005].

1.2 Programação Orientada a Linguagens

Programação Orientada a Linguagens (*LOP*, do inglês *Language Oriented Programming*) foi introduzida por Ward [1994] como sendo uma nova maneira de se organizar o desenvolvimento de grandes sistemas de software. Fowler descreve LOP como sendo uma maneira de descrever um sistema por meio de DSLs [Fowler, 2005]. A metodologia de LOP sugere que um ganho adicional de produtividade pode ser alcançado pelo desenvolvimento de uma linguagem orientada ao problema a ser resolvido, que depois é usada na implementação do sistema [Dmitriev, 2004].

Ward [1994] discute os problemas encontrados na produção de softwares em larga escala e os benefícios do uso de LOP. Ele advoga que o uso de softwares orientados

ao domínio da aplicação reduz o tempo de desenvolvimento, facilita a manutenção e o reúso. A abordagem se divide em três etapas:

1. Isolar um problema importante dentre as tarefas a serem realizadas pelo sistema. Definir, formalmente, uma nova linguagem de programação, especificamente orientada ao domínio do problema escolhido. Essa linguagem deve ter nível bem alto de abstração, e oferecer recursos para agilizar a especificação das soluções para esse problema.

2. Usando a linguagem construída, implementar as partes do sistema que estejam relacionadas ao problema escolhido.

3. Implementar um compilador ou interpretador para a nova linguagem.

Os passos descritos acima devem ser executados para quantos problemas se considerar adequados. O código final do sistema utilizará linguagens de programação convencionais e uma ou mais linguagens de domínio específico.

Na metodologia de programação orientada a linguagens, a etapa de construção das novas DSLs é considerada como um "nível médio" no processo de desenvolvimento. Isso possibilita atacar os problemas mais diretamente, e então trabalhar simultaneamente em direção a níveis mais baixos e níveis mais altos de abstração.

O uso adequado de técnicas de LOP no desenvolvimento de sistemas orientados a aspectos sugere que haverá um ganho de produtividade. Assim, faz-se necessário o desenvolvimento de sistemas que permitam ao programador aplicar essas técnicas.

A linguagem XAJ desenvolvida neste trabalho permite que a técnica de LOP possa ser usada em conjunto com programação orientada a aspectos.

1.3 Contribuições

A linguagem XAJ permite que sua sintaxe concreta seja estendida, possibilitando criar extensões e DSALs embarcadas de maneira modular. A definição da sintaxe e semântica das novas construções são dadas na própria linguagem proposta, sendo uma vantagem em relação a outras abordagens para criar extensões para linguagens. O "projetista de linguagens" não necessita utilizar diferentes métodos e ferramentas, o que poderia gerar problemas futuros, caso as tecnologias usadas não fossem estáveis, aumentando a portabilidade do código.

A linguagem proposta tem como principal objetivo permitir que novas extensões e linguagens de domínio específico orientadas a aspectos sejam definidas e utilizadas de

forma fácil, possibilitando que os usuários tirem proveito das vantagens proporcionadas pelo uso de LOP no desenvolvimento de sistemas.

Este trabalho traz como principal contribuição o desenvolvimento de uma linguagem extensível que permite usar a técnica de programação orientada a linguagens e permite que especificações das extensões sejam dadas de forma modular e portável. As contribuições deste trabalho estão listadas abaixo:

- Desenvolvimento de uma linguagem extensível orientada a aspectos;
- Implementação de um compilador para a linguagem desenvolvida;
- Modularização das extensões via classes sintáticas;
- Portabilidade das DSALs embarcadas na linguagem, pois são definidas na linguagem que é portável e não depende de ferramentas específicas para implementá-las;
- Mecanismo de importação de sintaxe, permitindo que extensões sejam usadas em pontos específicos;
- Mecanismo de importação de símbolos e uso, facilitando a definição de novas extensões baseadas na definição de outras já existentes;
- Definição do *using*, de modo que novas extensões podem ser definidas usando outras de maneira mais sucinta e não afeta globalmente a gramática da linguagem base.

1.4 Estrutura do Documento

O Capítulo 2 apresenta a linguagem AspectJ e as diversas abordagens para estendê-la. Faz uma revisão sobre linguagens existentes que permitem estender sua própria sintaxe concreta. A linguagem XAJ é apresentada no Capítulo 3 e a implementação do conjunto básico de funcionalidades da mesma é descrita no Capítulo 4. No Capítulo 5, alguns exemplos de extensões e DSAL são implementados usando a linguagem XAJ. As conclusões e trabalhos futuros são discutidos no Capítulo 6.

Capítulo 2

Extensibilidade de Linguagens Orientadas a Aspectos

Nos últimos anos, linguagens de domínio específico têm ganhado força como uma metodologia para desenvolvimento de softwares, visando ganho em produtividade e manutenção, e linguagens extensíveis aparecem como uma ferramenta para a implementação de DSLs. Porém, ainda não se conhece trabalhos de linguagens extensíveis orientada a aspectos, permitindo o desenvolvimento de DSALs embarcadas e extensões para tais linguagens.

Neste capítulo, uma introdução aos elementos básicos da linguagem AspectJ e uma revisão da literatura acerca das linguagens que permitem estender sua sintaxe são apresentadas. Por fim, as abordagens existentes para estender a linguagem AspectJ e as conclusões do capítulo são discutidas.

2.1 A Linguagem AspectJ

Programação orientada a aspectos ganhou popularidade rapidamente e a linguagem AspectJ [Laddad, 2003; Kiczales et al., 2001], uma extensão da linguagem Java com suporte a programação orientada a aspectos, é a principal linguagem orientada a aspectos em uso atualmente.

Um programa em AspectJ é constituído por classes Java e *aspectos*, os quais monitoram o programa em tempo de execução, podendo modificar o seu comportamento. O processo de modificar o programa a partir das informações dos aspectos é denominada de *combinação*.

As construções da linguagem AspectJ podem ser divididas em duas categorias, a *transversalidade estática* e a *transversalidade dinâmica*. A transversalidade estática

da linguagem é implementada a partir da análise estática do programa, ao passo que a transversalidade dinâmica está relacionada com a estrutura dinâmica do programa em tempo de execução.

2.1.1 Transversalidade Estática

Transversalidade estática da linguagem AspectJ permite modificar a estrutura estática do programa, e.g., classes, interfaces e aspectos do sistema, mas não modifica o comportamento dinâmico do sistema. As seguintes alterações podem ser feitas no sistema usando a linguagem AspectJ:

- introdução de campos e métodos em classes e interfaces;
- modificação da hierarquia de tipos;
- declarações de erros e advertências de compilação;
- enfraquecimento de exceções.

A implementação orientada a aspectos do padrão *visitor* para expressões aritméticas apresentada na Figura 1.3 utiliza transversalidade estática da linguagem para inserir o método *accept* nas classes *Mult, Number e Sum* e modifica a hierarquia de tipos da classe *Expr*.

2.1.2 Transversalidade Dinâmica

Transversalidade dinâmica consiste em introduzir um novo comportamento na execução de pontos específicos de um programa, modificando o fluxo de execução do sistema. Os recursos oferecidos pela linguagem para capturar os pontos em que novos comportamentos devem ser inseridos são denominados de *pontos de junção* e *conjuntos de junção*. Novos comportamentos são especificados usando os *adendos*.

2.1.2.1 Pontos de Junção

Pontos de junção são eventos bem definidos na execução do programa, e.g., chamadas de métodos ou execução de blocos de tratamento de exceções. Tais eventos podem ser capturados e adicionado novos comportamentos. Pontos de junção podem também possuir contexto associado. A Tabela 2.1 lista os pontos de junção disponíveis na linguagem AspectJ e descreve o contexto exposto.

Pontos de Junção	Contexto Exposto
Chamada de método	Chamador, objeto alvo, valor retornado, argumentos do método
Execução de método	Instância, argumentos do método, valor de retorno
Chamada de construtor	Chamador, argumentos do construtor
Execução de construtor	Objeto construído, argumentos do construtor
Execução de inicialização estática	Nada
Leitura de atributos	Instância referenciada, objeto alvo, valor do atributo
Escrita de atributos	Instância do valor a ser escrito, objeto alvo, valor de escrita
Execução de adendo	Aspecto, argumentos do adendo, valor de retorno do adendo
Pré-inicialização de objeto	Argumentos do construtor
Inicialização de objeto	Instância, argumentos do construtor
Execução de tratador de exceção	Instância, exceção capturada

Tabela 2.1. Pontos de Junção em AspectJ

2.1.2.2 Conjuntos de Junção

Conjuntos de junção são formados por um conjunto de pontos de junção e as informações de contexto destes pontos. Eles permitem a especificação de coleções de pontos de junção e a exposição de seus contextos para a implementação de adendos. As expressões de conjuntos de junção são formadas a partir de primitivas de conjuntos de junção que podem ser combinadas com os operadores lógicos *e (&&)*, *ou (||)* e *negação (!)*. A Tabela 2.2 contém as primitivas de conjuntos de junção da linguagem AspectJ.

2.1.2.3 Adendos

Adendos são definidos pelo programador e são executados para adicionar ou modificar o comportamento dos pontos de junção capturados. Os adendos podem ser de três tipos:

- O adendo *before* é executado antes da execução de um ponto de junção;
- O adendo *after* é executado depois da execução de um ponto de junção. Existem duas especializações deste adendo, que são:
  - O adendo *after returning* que é executado se o ponto de junção finaliza sua execução sem lançar exceção;

Primitivas de Conjunto de Junção	**Pontos de Junção Capturados**
execution	Execução de métodos e construtores
call	Chamadas de métodos e construtores
staticinitialization	Execução de inicialização estática
get	Leitura de atributos
set	Escrita de atributos
handler	Tratadores de exceção
initialization	Inicialização de objetos, parte depois da chamada ao método *super*
preinitialization	Inicialização de objetos, parte antes da chamada ao método *super*
adviceexecution	Execução de adendos
cflow	pontos de junção que ocorrem no fluxo de controle dos pontos de junção especificados no conjunto de junção
cflowbelow	Equivalente ao *cflow*, excluindo os pontos de junção do conjunto de junção especificado
within	Pontos de junção dentro das classes especificadas
withincode	Pontos de junção dentro dos métodos especificados
this	Pontos de junção com um valor de contexto *this* do tipo especificado
target	Pontos de junção em que o alvo no contexto tem o valor do tipo especificado
args	Pontos de junção com argumentos dos tipos especificados
if	Pontos de junção no qual um expressão é avaliada como *verdadeira*

Tabela 2.2. Primitivas de Conjuto de Junção

  - O adendo *after throwing* que é executado quando o ponto de junção lança uma exceção. Este adendo pode capturar a exceção;

- O adendo *around* é executado no lugar do ponto de junção. Opcionalmente, este adendo pode invocar o ponto de junção original.

2.2 Linguagens Extensíveis

Linguagens extensíveis são linguagens de programação que permitem ao programador adicionar ou modificar a sintaxe da linguagem base, assim como especificar sua semân-

tica. Extensibilidade de linguagens foi largamente discutida nas décadas de 60 e 70, tendo seu ápice com os simpósios de 69 e 71 [Christensen & Shaw, 1969; Schuman, 1971] e, nos últimos anos, o interesse da comunidade científica e da indústria em linguagens extensíveis tem crescido.

Standish [1975] faz um resumo acerca de linguagens extensíveis e discute quais características dessas linguagem tiveram mais sucesso. Standish classifica extensibilidade em três categorias: *paraphrase*, *orthophrase* e *metaphrase*. *Paraphrase* se refere a adicionar novas características à linguagem cujas semânticas são dadas a partir de construções já existentes na linguagem, e.g., sistema de macro, definição de novos tipos de dados. Extensibilidade por *orthophrase* significa adicionar novas características à linguagem nas quais a semântica não pode ser expressa em termos das construções existentes da linguagem base, e.g., adicionar sistema de entrada/saída a uma linguagem que não possua. *Metaphrase* consiste em modificar o significado de construções existentes da linguagem base.

A filosofia de linguagens extensíveis, na decáda de 60, pregava a possibilidade de o usuário customizar a linguagem de acordo com seus interesses. No entanto, extensões para linguagens, principalmente as *orthophrase* e *metaphrase*, são complexas e requerem um conhecimento considerável do usuário para sua implementação. Segundo Standish, linguagens extensíveis fracassaram porque os usuários comuns não são as pessoas mais adequadas para realizarem essa tarefa. Por outro lado, as extensões do tipo *paraphrase* tiveram mais sucesso por serem fáceis de implementar.

Baseado em experiências na indústria [van Deursen & Klint, 1997; Kieburtz et al., 1996], o uso de DSLs facilita a manutenção e desenvolvimento de sistemas, como foi discutido no Capítulo 1. Porém, um dos problemas associados ao uso de DSLs, como citado por van Deursen & Klint [1997], é o desenvolvimento e manutenção do compilador para a DSL desenvolvida. Assim, embora o custo de manutenção do sistema usando DSLs seja reduzido, deve-se manter o compilador para a nova linguagem criada. Neste contexto, linguagens extensíveis ressurgem como uma alternativa para implementar DSLs, e ainda, a responsabilidade da manutenção do compilador da linguagem extensível não será dos desenvolvedores da DSL. Além disso, linguagens extensíveis são uma alternativa para os sistemas que implementam as características que Deursen sugere que as ferramentas para desenvolver DSLs precisam ter, tal como criar DSLs de forma modular, com verificação estática do código escrito na DSLs e emitir messagens de erro apropriadas, dentre outras.

Em 1998, na conferência de OOPSLA, Programação Orientada a Objetos, Sistemas, Linguagens e Aplicações [Haungs, 1998], o *keynote* Guy Steele, na sua influente palestra intitulada *Growing a Language* [Steele, 1998], advoga que as linguagens de

programação devem ser pequenas e projetadas para crescerem. Segundo Steele, linguagens que são projetadas para atenderem as necessidades da maioria dos usuários serão extensas e, portanto, difíceis de aprender, sendo evitadas pelos usuários. Por outro lado, linguagens pequenas são fáceis de aprender, mas não atendem todas as necessidades dos usuários. Assim, as linguagens devem ser pequenas, pois facilitam o aprendizado e projetadas para crescerem de modo que os usuários vão aprendendo à medida que a linguagem cresce.

As motivações para linguagens extensíveis continuam válidas [Cheatham, 1969; Sammet, 1971; Schuman & Jorrand, 1970], porém a filosofia agora é deixar as empresas customizarem a linguagem, embutindo DSLs, de acordo com as suas necessidades e não os usuários com pouco conhecimento como foi imaginado nas décadas de 60 e 70.

2.2.1 Características de Linguagens Extensíveis

Uma linguagem extensível é composta pela linguagem base e características que permitem ao programador adicionar novas construções para a linguagem base ou modificar o comportamento de construções existentes [Schuman & Jorrand, 1970]. Para estender a linguagem base, é desejável que a linguagem extensível ofereça recursos, tais como [Duby, 1971; Clark., 2009]:

- Diagnóstico sintático preciso;
- Especificação modular das extensões;
- Definições de novas extensões não afetem as existentes;
- Novas características adicionadas à linguagem devem ser usadas como se fossem da linguagem base.

Na Seção 2.2.2, algumas linguagens extensíveis são apresentadas e discutidas em relação às características mencionadas, assim como as abordagens e dificuldades para implementá-las.

2.2.2 Exemplos de Linguagens Extensíveis

A maioria das linguagens que permitem estender a sua sintaxe utiliza o mecanismo de macro. Uma macro consiste em uma instrução que substitui certas partes do texto do programa fonte por novo texto e são implementadas, usualmente, por meio de um pré-processador para o compilador da linguagem. Um sistema de expansão de macros ingênuo, que simplesmente susbstitui o código da extensão por código equivalente na

linguagem base, pode gerar diversos problemas de amarração de variáveis. Um sistema de expansão de macros que não cria problemas de amarração é dito higiênico.

As principais linguagens que utilizam o sistema de macros para estender a sua sintaxe são as linguagens da familia de Commom Lisp [Steele Jr., 1990], como Scheme [Dybvig, 2009]. O mecanismo de macros oferecido pelas linguagens Scheme e Lisp é uma ferramenta poderosa para estender a linguagem e as extensões são usadas como se fossem da linguagem base, além de possuir um sistema de expansão de macros higiênico [Kohlbecker et al., 1986; Clinger & Rees, 1991].

XMF [Clark., 2009] é uma linguagem extensível que permite definir as extensões de maneira modular, por meio de *classes sintáticas* que definem a sintaxe e semântica das extensões. XMF permite a integração entre as novas extensões e a linguagem base, no entanto as novas construções devem ser usadas precedidas por um caractere de escape. A semântica das extensões em XMF é definida em termos da sintaxe base da linguagem, fazendo transformações na árvore de sintaxe abstrata. Um mecanismo de *quasi-quote* [Sheard & Jones, 2002; Huang et al., 2008] é usado para construir a árvore de sintaxe abstrata em termos da sintaxe concreta da linguagem base.

As informações de sintaxe da extensão definida na classe sintática geram uma tabela sintática LL(1), que pode conter conflitos e, neste caso, uma pilha de escolha é mantida de modo que o algoritmo possa retornar ao estado anterior caso seja feita a escolha errada. Depois de ter sido importada usando a diretiva *parserImport*, a extensão pode ser usada no programa depois do caractere de escape. Quando o caractere de escape é encontrado no programa, o analisador sintático muda a tabela de *parse* e faz a análise sintática da extensão encontrada e retorna à tabela de *parse* original quando a análise sintática da extensão termina. Essa abordagem permite que extensões sejam definidas em XMF sem que uma afete as outras.

XJ [Clark et al., 2008] é uma proposta de linguagem Java extensível, baseada no modelo de XMF. A linguagem XAJ desenvolvida neste trabalho foi inspirada na linguagem XJ, assim apresenta o mesmo conceito de classes sintáticas de XMF e XJ. Porém, as extensões definidas na linguagem XAJ são usadas como se fossem elementos da linguagem base, não sendo necessário o uso de um caractere de escape. XAJ apresenta um mecanismo de importação da sintaxe similiar ao *parserImport* de XMF, e ainda apresenta um mecanismo de importação de símbolos para aumentar a modularização na definição entre extensões definidas nas classes sintáticas. Todos os recursos da linguagem XAJ são discutidos em detalhes no Capítulo 3.

O modelo para definir a semântica das extensões usada pelas linguagens XMF e XAJ é similar ao processo de macros, porém opera na árvore de sintaxe abstrata e, teoricamente, pode usar informações de contexto. Modificar a árvore de sintaxe

abstrata também pode gerar os mesmos problema de amarração de variáveis, não sendo higiênica.

Schuman [Schuman & Jorrand, 1970] apresenta um modelo para estender a sintaxe de linguagens por meio da modificação da gramática da linguagem. Schuman define três componentes em seu modelo, Φ (produções), π (predicado) e ρ (substituição). O primeiro componente são as produções que devem ser inseridas na gramática da linguagem base, sendo obrigatória na declaração da extensão, os outros dois componentes são opcionais e representam condições adicionais que devem ser satisfeitas e código que traduz a extensão para construções da linguagem base. O mecanismo de extensibilidade das linguagens XMF e XAJ, via classes sintáticas, é parecido com o modelo proposto por Schuman.

A linguagem Fortress [Allen et al., 2008] foi desenvolvida para aplicações científicas e computação de alto desempenho. Fortress tem recursos para paralelismo implícito, transações, sintaxe que emula notação matemática e permite estender a sua sintaxe concreta através de macros. O sistema de extensibilidade de Fortress foi projetado para atender às seguintes metas [Allen et al., 2009]:

- Nova sintaxe deve ser indistinguível da antiga, do ponto de vista do usuário;
- As macros devem ser verificadas se são bem formadas antes de serem expandidas;
- A definição e uso das macros devem ser encapsuladas (macros higiênicas e composição de macros);
- Suportar definição de macros recursivamente.

Allen et al. [2009] discutem as dificuldades de realizar a análise sintática da linguagem Fortress e a abordagem adotada para atender as metas do projeto. Segundo Allen, uma das maiores dificuldades de realizar a análise sintática da linguagem se dá pelo fato da nova sintaxe ser indistinguível da original, aliado ao suporte à notação matemática. Se fosse usado um caractere de escape antes do uso das extensões, como usada em XMF, a análise sintática seria fácil, porém a sintaxe da extensão não seria indistinguível da sintaxe da linguagem base, não atendendo uma das metas do projeto.

Extensões em Fortress são feitas por meio de definições de gramáticas, que podem ser combinadas com outras de forma modular. Uma gramática define um conjunto de não-terminais, extensões e pode estender outras gramáticas. A semântica da nova extensão é especificada por uma regra de transformação que traduz a extensão em termos da sintaxe base da linguagem Fortress. Macros também podem ser usadas para especificar a semântica da nova construção.

Um programa em Fortress é composto por uma sequência de definições de gramáticas (extensões ou macros) e uma expressão principal. A análise sintática do código é feita em duas etapas: *parsing* e *transformation*.

Na etapa de *parsing*, a AST representando o programa é construída. Nesta etapa, primeiramente é feita a análise sintática das definições de gramáticas, pois essas definem como é feita a análise sintática das extensões que podem aparecer na expressão principal e nas regras de transformação. A expressão principal e as regras de transformação são representadas na AST como cadeias de caracteres. Na segunda etapa é realizada a expansão das macros, transformando o programa para a sintaxe base de Fortress.

Antes de realizar a análise sintática da expressão principal e das regras de transformação, o compilador de Fortress cria PEGs para as definições de gramática. PEG [Ford, 2004] (*do inglês, Parsing Expression Grammar*) é usada para definir gramáticas, assim como o formalismo de gramáticas livres de contexto (GLC). As gramáticas construídas não são ambíguas, são fechadas sobre a operação de união e integram a análise léxica com a sintática, por isso foram usadas no desenvolvimento do compilador para Fortress, segundo os autores. Um analisador sintático é gerado automaticamente usando o Rat! [Grimm, 2006] depois que as PEGs são contruídas, e usado para realizar a análise sintática da expressão principal e das regras de transformação. Depois de realizada a análise sintática, as extensões na árvore de sintaxe abstrata são substituídas por nós que contém somente construção da linguagem base, de forma higiênica.

O mecanismo para estender a sintaxe concreta usado por Fortress, assim como a abordagem usada na implementação do compilador para a linguagem é similar ao usado pela linguagem XAJ (para detalhes da implementação do compilador de XAJ leia o Capítulo 4). A principal diferença no mecanismo de extensibilidade usado por XAJ é que as extensões não têm escopo local como em Fortress e podem ser reutilizadas por outros programas.

π [Knöll & Mezini, 2009] é uma nova linguagem de programação que provê um mecanismo de abstração baseado em parametrização de símbolos. A linguagem tem somente uma construção, *padrão*, e programar em π é basicamente construir os padrões e avaliá-los. A linguagem π pode ser considerada como um novo paradigma de programação, uma linguagem de padrões. Extensões para a linguagem podem ser enxergadas como construções de novos padrões para o conjunto de padrões básico da linguagem.

Camlp5 [de Rauglaudre, 2009] é um pré-processador para a linguagem OCaml que realiza transformações no código, assim como macros. Camlp5 permite adicionar, modificar ou eliminar regras da gramática da linguagem OCaml. Assim como as linguagens Fortress, XMF e XAJ, em Camlp5, a semântica das construções é dada modificando a árvore de sintaxe abstrata, de modo que os nós das extensões são subs-

tituidos por elementos da linguagem base.

Java Syntactic Extender [Bachrach & Playford, 2001] é um pré-processador, assim com Camlp5, porém para a linguagem Java. Java Syntactic Extender usa *Skeleton syntax tree (SST)*, uma biblioteca que provê representação de código, para fazer transformações no código. O Java Syntactic Extender cria a SST para o código do programa e depois as macros são expandidas top-down, recursivamente.

Assim como Java Syntactic Extender e Camlp5, existem diversos sistemas que oferecem recursos para estender a sintaxe das linguagens ou implementar linguagens de domínio específico. Dentre eles, pode-se destacar:

- Jakarta Tool Suite (JTS) [Batory et al., 1998] - um sistema para implementar DSLs embutidas na linguagem Java;

- OpenJava [Tatsubori et al., 2000] - suporta o *meta object protocol (MOP)* para a linguagem Java, permitindo criar algumas extensões para a linguagem, porém não inclui mecanismo para estender a sintaxe concreta de Java;

- MetaBorg [Bravenboer et al., 2006a; Riehl, 2006; Bravenboer & Visser, 2004; Bravenboer et al., 2008] - método para embutir DSLs em uma linguagem hospedeira, usando uma definição modular da sintaxe da linguagem com o conjunto de ferramentas SDF [Heering et al., 1989] e Stratego/XT Visser [2001];

- Silver [Wyk et al., 2008] - um arcabouço para descrição de linguagens baseado em gramáticas de atributos projetado para a composição e construção modular de linguagens de domínio específico;

- Maya [Baker & Hsieh, 2002] - permite definir extensões para a linguagem Java usando um mecanismo de casamento de padrões para expandir macros na árvore de sintaxe abstrata, porém as extensões têm efeito global e não modularizadas;

- Lua [Ierusalimschy et al., 1996] - uma linguagem de extensão embutida em linguagens de propósito geral que contém um mecanismo de *fallback* que permite estender a semântica de algumas construções da linguagem, e.g., operadores. Porém, a linguagem não tem mecanismo para estender a sintaxe concreta.

2.3 Abordagens Para Extensões de Linguagens Orientadas a Aspectos

Linguagens extensíveis são uma boa solução para oferecer suporte à criação de DSLs embutidas, no entanto não se tem conhecimento de nenhuma linguagem extensível que seja orientada a aspectos que possibilite desenvolver extensões para tais linguagens. Para esta finalidade, deve-se usar outras alternativas, e.g., arcabouços ou construir um compilador especializado. Nesta seção, serão descritos alguns desses sistemas.

2.3.1 Arcabouços Para Estender Linguagens Orientadas a Aspectos

Meta-AspectJ (MAJ) [Huang & Smaragdakis, 2006; Huang et al., 2008] é uma extensão de Java que permite criar programas que geram código em AspectJ. MAJ pode ser usada para implementar linguagens de domínio específico orientadas a aspectos, gerando código das DSALs na linguagem AspectJ, usando anotações. Porém, não existe nenhum mecanismo para estender a sintaxe concreta.

XAspects [Shonle et al., 2003] define um mecanismo baseado em plugin para desenvolver linguagens de domínio específico orientada a aspectos. O único ponto para estender a sintaxe concreta é a definição de aspectos, sendo uma abordagem muito restritiva para extensibilidade de linguagens. As extensões definidas em XAspects são modulares, porém não pode ser considerada portável, pois são dependentes do mecanismo de puglin.

Josh [Chiba & Nakagawa, 2004] permite definir novos pontos de junção para um subconjunto da linguagem AspectJ com separação distinta dos processos de tempo de execução e tempo de costura. Caso seja necessário algum teste em tempo de execução, deve ser inserido explicitamente usando um arcabouço para manipulação de *bytecode*. Assim como Josh, Join Point Selectors [Breuel & Reverbel, 2007] é um método que permite criar novos pontos de junção mais expressivos que os oferecidos pela linguagem AspectJ.

Bagge & Kalleberg [2006] consideram linguagens de domínio específico orientadas a aspectos uma abstração sintática para bibliotecas de transformação, do mesmo modo que linguagens de domínio específico são abstrações sintáticas para uma biblioteca da linguagem base em questão. A ferramenta de transformação Stratego/XT [Bravenboer et al., 2008] é utilizada para implementar as características transversais das linguagens orientadas a aspectos. Assim como XAspects, esta estratégia também não é portável, pois é dependente da ferramenta de transformação de código.

Nos trabalhos de Wyk [2003, 2007], algumas características da linguagem AspectJ, i.e., aspectos e o processo de costura, são implementadas como extensões para uma linguagem base, orientada a objetos, via gramáticas de atributos [Vogt et al., 1989], usando um mecanismo denominado *forwarding* [Wyk et al., 2002]. Extensões para linguagens orientadas a aspectos podem ser implementadas usando esta técnica.

2.3.2 Implementações Costumizadas de Compiladores Para Linguagens Orientadas a Aspectos

Uma alternativa para adicionar novas extensões às linguagens orientadas a aspectos é construir um novo compilador para a linguagem ou modificar um existente. Existem dois compiladores para a linguagem AspectJ que podem ser usados para tal finalidade, o *ajc* [AspectJ, a] e o *abc* [Avgustinov et al., 2005]. O *ajc* é o compilador oficial da linguagem desenvolvido para fazer uma compilação incremental e eficiente, enquanto que o *abc* foi projetado para ser extensível e testar novas funcionalidades para a linguagem e otimizações de código. No entanto, customizar um compilador é uma tarefa custosa e requer um conhecimento do compilador, mesmo usando um compilador extensível, como o *abc*.

Bravenboer et al. [2006b] desenvolveram um *front-end* para a linguagem AspectJ usando o SDF [Heering et al., 1989], possibilitando que extensões para a linguagem sejam definidas de forma modular. A análise sintática é feita usando um analisador sintático gerado automaticamente (Scannerless Generalized-LR Parsing [Visser, 1997a]), porém não existe um *back-end* integrado a esta ferramenta. Esta abordagem é útil para fazer protótipos e testar novas construções para a linguagem AspectJ, mas depois deve ser implementado um compilador para a linguagem com as novas construções ou implementá-la em uma linguagem extensível.

2.4 Conclusão

Neste capítulo, uma revisão da literatura sobre linguagens extensíveis foi apresentada. Como discutido ao longo do capítulo, linguagens extensíveis já foram amplamente discutidas nas décadas de 60 e 70, e nos últimos anos, diversas pesquisas têm sido realizadas nesta área, tendo como principal motivação a customização das linguagens para embutir linguagens de domínio específico, visando uma ganho adicional em manutenibilidade e produtividade no desenvolvimento de softwares.

Existem diversas ferramentas que possibilitam às empresas embutirem linguagens de domínio específico em uma linguagem hospedeira. Porém, a principal desvantagem

desta abordagem é que as empresas serão as responsáveis por manterem o compilador para a linguagem e muitas dessas ferramentas não oferecem os recursos citados na Seção 2.2.1, além de não oferecerem portabilidade para as extensões. Assim, as linguagens extensíveis aparecem como uma alternativa para implementar extensões, oferecendo as seguintes vantagens:

- Podem oferecer todos os recursos citados na Seção 2.2.1, tais como: desenvolvimento modular das extensões, integração com a linguagem base, diagnóstico de erro preciso;
- Manutenção da linguagem não será responsabilidade do desenvolvedor da extensão;
- Possibilidade de integração com ferramenta de desenvolvimento, e.g., Eclipse;
- Portabilidade das extensões.

Apesar de linguagens extensíveis não serem novidade, ainda há muito a evoluir, e pesquisas precisam ser feitas nesta área. Além disso, a introdução desses recursos em linguagens orientada a aspectos é assunto atual de pesquisa. XAJ, uma extensão da linguagem AspectJ desenvolvida neste trabalho, é a primeira linguagem extensível orientada a aspectos, que se tem conhecimento. Os próximos capítulos tratarão da linguagem XAJ.

Capítulo 3

A Linguagem XAJ

A linguagem XAJ foi inicialmente proposta por Di Iorio et al. [2009], apresentando exemplos significativos de sua utilização. Em 2009 [Reis et al., 2009] uma primeira versão da linguagem e um compilador foram apresentados. No decorrer deste trabalho algumas melhorias foram acrescentadas à proposta original da linguagem. Neste capítulo, será apresentada a definição formal da gramática de XAJ. Sua semântica será apresentada informalmente.

A linguagem XAJ *(eXtensible AspectJ)* é uma extensão da linguagem AspectJ que permite estender sua sintaxe concreta. XAJ apresenta o conceito de *classes sintáticas*, que são novas unidades de compilação que servem para criar extensões para a linguagem. As classes sintáticas possibilitam que a gramática da linguagem AspectJ seja modificada, acrescentando novas produções ou modificando o significado de produções existentes, e a semântica de novas construções é especificada por métodos especiais definidos na própria classe sintática. A finalidade de criar extensões e embarcá-las em uma linguagem de próposito geral é oferecer recursos mais apropriados à linguagem para expressar um problema em questão. Com esse propósito, XAJ contém somente funcionalidades para aumentar o conjunto de regras da linguagem AspectJ, não contendo recursos para eliminar produções da linguagem original.

As classes sintáticas servem também para construir a árvore de sintaxe abstrata, *AST, do inglês Abstract Syntax Tree*, representando as extensões na mesma, depois de realizada a análise sintática. As classes sintáticas têm as seguintes finalidades:

- Especificar a sintaxe das novas construções;
- Definir a semântica das novas construções ou modificar a semântica de construções existentes;

- Representar, na árvore de sintaxe abstrata, as novas construções.

As classes sintáticas aumentam a modularidade das novas construções no sentido de que toda a definição relativa à construção, i.e., sintaxe e semântica, está encapsulada em uma mesma unidade de compilação. XAJ também aumenta a portabilidade das extensões, pois elas são definidas na própria linguagem, não dependendo de ferramentas específicas em sua implementação.

3.1 Classes Sintáticas

Classes sintáticas são novas unidades de compilação, assim como *classes* e *aspectos*, que servem para estender a própria linguagem XAJ, e são declaradas no programa usando a palavra-chave *syntaxclass*[1]. As extensões para a linguagem são criadas usando *declarações sintáticas*, as quais são especificadas no corpo de uma classe sintática. A semântica das novas construções é dada em um método especial da classe sintática, denominado *desugar*. Depois de a árvore de sintaxe abstrata ser construída, contendo nós de extensão representados pelas classes sintáticas, o método *desugar* é executado em tempo de compilação e os nós das novas construções são substituídos por uma sub-árvore cujos nós têm somente construções na linguagem AspectJ.

As classes sintáticas, tais como classes e aspectos, possuem modificadores de acesso *public* ou *private*, cuja semântica é idêntica à usada para classes e aspectos. Classes sintáticas podem conter funções e campos, do mesmo modo que classes e aspectos, além de uma declaração sintática. A Figura 3.1 apresenta uma classe sintática, de nome *MultiIntro*, para criar a extensão *MultiIntro* discutida na Seção 1.1. A classe sintática *MultiIntro* não apresenta campos, contém apenas o método *desugar* e a declaração sintática que define a sintaxe da extensão criada.

Uma declaração sintática é declarada usando o comando especial *@grammar*, como apresentado na Figura 3.1, e tem a semântica de modificar a gramática da linguagem XAJ, adicionando novas produções, novos não-terminais ou modificando o comportamento de produções existentes. Existem três tipos de declaração sintática:

- Declaração sintática simples;
- Declaração sintática estendida;
- Declaração sintática de sobrecarga.

[1] Inicialmente a palavra-chave *class* foi usada para declarar classes sintáticas também, mas por se tratar de uma nova unidade de compilação, usou-se *syntaxclass* para diferenciá-la das classes usuais

```
public syntaxclass MultiIntro {
  @grammar extends intertype_member_declaration {
    MultiIntro ->
     modifiers = modifiers_opt
     returnType = type
     className = identifier
     "+."  methodName = identifier
     "(" params = formal_parameter_list_opt ")"
     introducedCode = block;
  }
  public Node desugar(Context ctx, NodeFactory nf,
    TypeSystem ts) {
    ClassDecl cd = ctx.getClass(className);
    ClassDecl sub[] = cd.getSubClasses();
    List list = nf.TypedList(ClassMember.class);
    for(ClassDecl x :  sub)
      list.add(nf.IntertypeMethodDecl(modifiers,
        returntype, x, methodName, params, block));
     return list;
  }
}
```

Figura 3.1. Classe sintática para definir a extensão MultiIntro.

A Figura 3.1 apresenta um exemplo de uma *declaração sintática estendida*, caracterizada pela palavra-chave *extends* depois de *@grammar*. A sintaxe e semântica das declarações sintáticas são discutidas detalhadamente na Seção 3.2.

A semântica das extensões definidas pelas declarações sintáticas é dada no método *desugar*. Este método é executado em tempo de compilação e irá substituir o nó que representa a extensão por novas construções da linguagem AspectJ, usando programação generativa. No método *desugar* da Figura 3.1, uma declaração intertipo para cada subclasse de *classeName* é criada e adicionada em uma lista de nós que é retornada no fim do método.

Em algumas aplicações, pode ser necessário mais de um passo para traduzir uma extensão, ou informações devem ser colhidas antes da tradução ser realizada. Nesses casos específicos, uma versão alternativa do método *desugar* pode ser usada, o qual recebe um parâmetro adicional com o valor do passo atual. A versão do método *desugar*

com o parâmetro indicando o número do passo para a classe sintática *MultiIntro* seria:

```
public AST desugar(Context ctx, NodeFactory nf, TypeSystem ts,
                   int pass) {
  if (pass == 1) { ...  }
  else if (pass == 2) { ...  }
  ...
}
```

O número de passos que uma classe sintática necessita deve ser especificado na mesma, usando o comando *@numberOfPasses*. Se a classe sintática *MultiIntro* precisasse de dois passos, por exemplo, ela teria a seguinte declaração:

```
public syntaxclass MultiIntro {
  @numberOfPasses = 2;
  @grammar extends ...
}
```

Uma classe sintática tem uma única declaração sintática e no máximo um uso do comando *@numberOfPasses*. Se o comando *@numberOfPasses* não for usado, assume-se que a classe sintática necessita de um único passo e a versão alternativa do método *desugar* não será chamado.

A linguagem XAJ oferece duas formas para especificar o escopo das extensões definidas pelas classes sintáticas, a importação da sintaxe e a importação de símbolos, que funcionam de forma semelhante ao *import* da linguagem AspectJ. No primeiro caso, o escopo é definido pela palavra-chave *importsyntax* cuja semântica é especificar que a sintaxe da extensão definida pela classe sintática importada pode ser usada no escopo da importação. Um exemplo de importação sintática é apresentado na Figura 3.2, que mostra o código da implementação do padrão Visitor para o exemplo apresentado na Seção 1.1 que utiliza a extensão *MultiIntro*. A declaração *importsyntax MultiIntro;* significa que a sintaxe definida pela classe sintática *MultiIntro* pode ser usada nesse escopo.

A importação de símbolos é usada por classes sintáticas para importar novos não-terminais definidos por outras classes sintáticas, permitindo que classes sintáticas use em sua declaração sintática a definição da sintaxe de extensões definidas em outras classes sintáticas. A importação de símbolo só define que o nome deste símbolo pode ser usado dentro do escopo que foi importado. As Figuras 3.3 e 3.4 apresentam classes sintáticas que definem as extensões *ExemploA* e *ExemploB*, respectivamente. Na Fi-

```
importsyntax MultiIntro;

public aspect VisitorAspect {
  declare parents:  Expr extends ElementVisitor;
  void Expr+.accept(Visitor v) {
    v.visit(this);
  }
}
```

Figura 3.2. Aspecto que importa a sintaxe da extensão definida na classe sintática MultiIntro.

```
public syntaxclass ExemploA {
  @grammar {
    ExemploA -> ...
     Restante da sintaxe da extensão
}
```

Figura 3.3. Definição de uma extensão ExemploA.

```
importsymbol ExemploA;

public syntaxclass ExemploB {
  @grammar {
    ExemploB -> ...
     ExemploA
     ...
}
```

Figura 3.4. Exemplo de uso de importação de símbolo.

gura 3.3, o novo não-terminal *ExemploA* é criado e adicionado à gramática do XAJ, e na Figura 3.4, este não-terminal é importado, definindo que este nome pode ser usado dentro do escopo.

As importações de símbolos e sintaxe aumentam a modularidade das extensões. Na Seção 5.1, é apresentada uma versão mais complexa da extensão *MultiIntro* que mostra como a modularidade do código é aumentada usando os mecanismos de importação de símbolos e sintaxe da linguagem XAJ.

3.2 Gramática de XAJ

A linguagem XAJ é uma extensão da linguagem AspectJ, portanto a gramática da linguagem XAJ foi construída adaptando a gramática da linguagem AspectJ para per-

```
syntax_class = [modifiers] “syntaxclass” identifier
    [extends] [interfaces] syntax_class_body.
syntax_class_body = “{” [number_passes] syntax_declaration
    {class_member_declaration} “}”.
syntax_declaration = “@grammar” [using] “{” {production} “}”
    | “@grammar” “extends” non_terminal [using]
        “{” {production} “}”
    | “@grammar” “overrides” non_terminal “;”.
using = “using” non_terminal {“,” non_terminal}.
production = non_terminal “->” expression “;”.
expression = term [semantic_action]
    {“|” term [semantic_action]}.
term = factor {factor}.
factor = [identifier “=”] non_terminal
    | string_literal
    | [identifier “=”] “(” exp “)”
    | [identifier “=”] “[” exp “]”
    | [identifier “=”] “{” exp “}”.
exp = term {“|” term}.
non_terminal = identifier.
semantic_action = “{:”  {statement} “:}”.
number_passes = “@numberOfPasses” “=” integer_literal “;”.
```

Figura 3.5. Definição da gramática do XAJ.

mitir que classes sintáticas sejam definidas, incluindo todas as características discutidas na seção anterior. A Figura 3.5 apresenta a gramática para as classes sintáticas e seus elementos, i.e., declarações sintáticas e comando *@numberOfPasses*.

A primeira regra da gramática, apresentada na Figura 3.5, especifica a sintaxe das classes sintáticas, a qual têm opcionalmente um modificador de acesso, seguido pela palavra-chave *syntaxclass*, nome da classe sintática (ou nome da extensão), superclasse e interfaces opcionalmente e o corpo da classe sintática. A Figura 3.6 mostra uma declaração de classe sintática que estende uma invocação de método, significando que onde uma invocação de método é esperada, também poderá aparecer a extensão *ExCall*. Para representar este nó corretamente na AST, a classe sintática deste exemplo deve implementar a interface *Call* e estender a classe que contém as implementações padrão das operações de *Call* (*Call_c*). As definições de interfaces e superclasse da classe sintática são referente exclusivamente ao nó que ela representa na AST, não tendo nenhuma relação com herança de extensões. As interfaces e superclasse que a classe sintática devem implementar são geradas automaticamente pelo compilador da linguagem XAJ, quando não forem especificadas na definição da classe sintática.

Na Figura 3.7, dois exemplos típicos de um corpo de uma classe sintática são apresentados. Na parte *(a)* da figura, o corpo da classe sintática *ExBody* contém o

```
public syntaxclass ExCall extends Call_c implements Call {
  @grammar extends method_invocation { ...}
}
```

Figura 3.6. Exemplo de um cabeçalho de classe sintática.

```
public syntaxclass ExBody {
  @numberOfPasses = 2;
  @grammar { ...}
  public Node desugar( ...)  {
    ...
  }
}
```

(a) Classe sintática que define o número de passos

```
public syntaxclass ExBody {
  @grammar { ...}
  public Node desugar( ...)  {
    ...
  }
}
```

(b) Classe sintática que usa o número de passos padrão

Figura 3.7. Exemplo de corpo de classe sintática.

comando para definir o número de passos (*@numberOfPasses*) seguido por uma única declaração sintática e por membros de classes (não-terminal *class_ member_ declaration* da gramática original de AspectJ). Na parte *(b)* da figura, o corpo da classe sintática não contém o comando opcional que define o número de passos, sendo por padrão o valor um. A ordem da declaração sintática e do comando *@numberOfPasses* é importante, os quais devem ser especificados antes de qualquer declaração de membro de classe.

Uma declaração sintática, especificada pelas produções do não-terminal *syntax_ declaration* da gramática na Figura 3.5, é declarada usando a palavra-chave *@grammar*, que pode ser seguida pelas palavra-chaves *extends* (declaração sintática estendida) ou *overrides*(declaração sintática de sobrecarga). As declarações sintáticas simples (quando *@grammar* não é seguido por nenhuma palavra-chave) e a declaração sintática estendida têm a semântica de inserir as regras definidas no corpo da declaração na gramática da linguagem XAJ. As Figuras 3.3, 3.1 e 3.9 mostram classes sintáticas que usam declarações sintáticas simples, estendida e de sobrecarga, respectivamente.

Opcionalmente, as declarações de sintaxe simples e estendida podem conter *declarações de uso*, denotada pela palavra-chave *using*, gerada pelo não-terminal *using* da gramática da Figura 3.5. A semântica de uma declaração de uso especifica que as regras da extensão que aparecem na declaração são alcançáveis somente a partir de algum não-terminal definido pela declaração de sintaxe em questão. As declarações de uso facilitam na definição de novas extensões, como será mostrado no Capítulo 5, além de ter o efeito de modificar a gramática da linguagem localmente. A Figura 3.8 apre-

```
importsymbol ExCall;
public syntaxclass Using {
  @grammar using ExCall {
    Using → body;
  }
  ...
}
```

Figura 3.8. Exemplo de uma declaração de uso.

senta a definição de uma classe sintática que tem o mesmo comportamento de um bloco e tem uma declaração de uso da nova invocação de método definida pela classe sintática da Figura 3.6. Observe que a classe sintática *Using* não usa o símbolo *ExCall* no corpo de suas produções, porém a declaração de uso especifica que ela poderá ser alcançada, assim o usuário da extensão *Using* poderá usar o novo comportamento de chamada de método definido por *ExCall*, mas somente dentro do corpo gerado pelo não-terminal *body* (não-terminal da gramática original de AspectJ que produz um bloco de comandos) da extensão *Using*, não podendo ser usado dentro de um corpo de método, por exemplo. Para que a extensão *Using* tenha o mesmo comportamento definido sem usar a declaração de uso, a extensão teria que redefinir todas as regras que são geradas por *body*, modificando os locais onde pode ocorrer uma chamada de método para adicionar a nova possibilidade definida pela extensão *ExCall*, porém com a declaração de uso é economizado todo este esfoço.

A declaração sintática da classe sintática para a extensão *MultiIntro* discutida na Seção 1.1 e apresentada na Figura 3.1, acrescenta as seguintes regras à gramática da linguagem XAJ:

```
intertype_member_declaration → MultiIntro
MultiIntro → modifiers_opt type identifier "+."
    identifier "(" formal_parameter_list_opt ")" block
```

Os nomes *intertype_member_declaration* (declaração intertipo), *modifiers_opt* (modificadores de acesso opcional), *type* (tipo), *identifier* (identificador), *formal_parameter_list_opt* (lista de parâmetro opicional), *block* (bloco de comandos) são não-terminais da gramática AspectJ[2] e *MultiIntro* é um novo não-terminal definido pela declaração sintática. Neste exemplo é usada uma declaração sintática estendida, cuja diferença para a declaração sintática normal é que esta adiciona uma produção

[2]A gramática da linguagem Aspectj, assim como os símbolos usados nos exemplos estão nos apêndices B e C

```
public syntaxclass ExemploC {
  @grammar overrides intertype_member_declaration;
  public AST desugar(Context ctx, NodeFactory nf,
    TypeSystem ts) { ...}
}
```

Figura 3.9. Exemplo de uso da declaração sintática de sobrecarga.

de um não-terminal existente na gramática para o novo não-terminal definido pela primeira produção do corpo da declaração. Assim as produções adicionadas à gramática podem ser alcançadas pelo símbolo inicial da gramática original de AspectJ. No exemplo discutido, a seguinte regra é adicionada, especificando que a extensão *MultiInttro* pode ser usada em qualquer ponto onde uma declaração intertipo é esperada:

$$\texttt{intertype\_member\_declaration} \rightarrow \texttt{MultiIntro}$$

Uma declaração sintática de sobrecarga não adiciona nenhuma produção à gramática, mas sobrescreve o método *desugar* executado para um dado não-terminal. Na Figura 3.9, a declaração sintática de sobrecarga especifica que o método *desugar* desta classe sintática será executado para *intertype_ member_ declaration*.

As declarações sintáticas normal e estendida definem no mínimo uma nova produção (não-terminal *production*). Cada produção cria um novo não-terminal de nome único, diferente de qualquer não-terminal da gramática original. A produção principal é a primeira definida, e o nome do não-terminal deve ser igual ao nome da classe sintática na qual está sendo definida. Todas as outras produções são auxiliares, permitindo que construções mais complexas, e.g., recursividade, sejam definidas. As produções podem conter, opcionalmente, uma ação semântica (*semantic_ action*) cuja finalidade é exclusivamente criar o nó da AST que representa a extensão. Para isso, toda ação semântica deve ter um comando *new* para criar um objeto da classe sintática, que irá representar a extensão na AST e esse objeto deve ser atribuído à variável *RESULT*. Quando o usuário não especifica a ação semântica da produção, o compilador da linguagem XAJ gera o código necessário para criar este nó. A Figura 3.10 mostra um exemplo de ação semântica.

Os tipos de produções que podem ser adicionados à gramática da linguagem são regras de gramáticas livre de contexto com algumas facilidades para criar lista (repetição de zero ou mais símbolos entre { e }) e opcional (no máximo um símbolo entre [e]). Alguns termos de uma produção podem ter sinônimos (*named_ factor*), de modo que possam ser referenciados na ação semântica, na própria classe sintática

```
public syntaxclass ExAcao {
  @grammar {
    ExAcao → A1 A2 ... An;
      {:  RESULT = new ExAcao(); :}
  }
...
}
```

Figura 3.10. Exemplo de uma ação semântica.

```
type_declaration = syntax_class.
class_member_declaration = syntax_class.
```

Figura 3.11. Regras para adicionar as classes sintáticas a linguagem AspectJ.

```
import_declaration = import_syntax
    | import_symbol.
import_syntax = "importsyntax" qualified_name [".*"] ";"
import_symbol = "importsymbol" qualified_name [".*"] ";"
```

Figura 3.12. Regras para adicionar as importações de sintaxe e símbolos à linguagem AspectJ.

ou pelo método *desugar*. Métodos *get* e *set* para os sinônimos definidos são gerados automaticamente, podendo ser acessados por outras entidades. A Figura 3.13 mostra como fica a classe sintática *MultiIntro* depois que os campos e métodos *get* e *set* para os sinônimos forem gerados[3]. Os tipos dos atributos gerados automaticamente são determinados a partir de um conjunto de classes que representam os não-terminais da gramática na árvore de sintaxe abstrata.

As produções da Figura 3.11 foram adicionadas à gramática da linguagem AspectJ para permitir que classes sintáticas sejam declaradas como tipo (*type_ declaration*) e como membro de classes (*class_ member_ declaration*), assim uma classe sintática pode aparecer nos mesmos locais que classes ou aspectos podem ser declarados. As importações de sintaxe e símbolos podem ser declaradas em qualquer lugar que uma importação comum seja aceita. A Figura 3.12 mostra como as importações foram inseridas à gramática do AspectJ.

Os símbolos *import_ declaration* (declaração de importação) e *qualified_ name* (nome qualificado) são originais da gramática AspectJ. Uma importação de sintaxe ou de símbolo pode importar todos os elementos de um pacote ou de uma única classe

[3]Por questões de espaço, alguns métodos e atributos gerados automaticamente foram omitidos

```
public syntaxclass MultiIntro {
  @grammar extends intertype_member_declaration {
    MultiIntro ->
     modifiers = modifiers_opt
     returnType = type
     className = identifier
     "+."  methodName = identifier
     "(" params = formal_parameter_list_opt ")"
     introducedCode = block;
  }
  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    ClassDecl cd = ctx.getClass(className);
    ClassDecl sub[] = cd.getSubClasses();
    List list = nf.TypedList(ClassMember.class);
    for(ClassDecl x :  sub)
      list.add(nf.IntertypeMethodDecl(modifiers,
        returntype, x, methodName, params, block));
     return list;
  }
  public MultiIntro(Flags modifiers, ...)  { ...}
  private Flags modifiers;
  public Flags getmodifiers() { return modifiers; }
  public void setmodifiers(Flags modifiers) {
    this.modifiers = modifiers;
  }
  ...
}
```

Figura 3.13. Classe sintática MultiIntro com membros gerados automaticamente

sintática.

3.3 Método *desugar*

A gramática da linguagem não força a declaração do método *desugar*, isto é feito quando são inseridas automaticamente na classe sintática as interfaces que ela deve implementar. O método *desugar* é executado pelo compilador de XAJ depois que a AST foi construída, a qual contém nós que representa as extensões, e tem como função substituir esses nós por uma subárvore que contém somente nós da linguagem AspectJ. Para executar essa operação, é passado como parâmetro para o *desugar* um objeto de contexto que oferece algumas operações básicas sobre a AST, tais como verificar se existe uma classe dado o seu nome, retornar todas as subclasses de uma dada classe. O método recebe também uma fábrica de nós da AST para construir a subárvore que será retornada e um sistema de tipos, que define os tipos da linguagem base, sendo usado

para suprir alguns parâmetros de métodos da fábrica, em alguns casos. Em versões futuras da linguagem, será adicionado o mecanismo de *quasi-quote* usando programação generativa, evitando que o usuário tenha que fazer chamadas explicitas à fábrica para criar os nó da AST.

Os nós que representam os elementos da linguagem AspectJ na AST são os usados pelo compilador do *abc*, assim a linguagem está de certa forma dependente dessa implementação. Porém, com o mecanismo de *quasi-quote*, essa dependência pode ser contornada.

3.4 Conclusão

A linguagem XAJ, uma extensão da linguagem AspectJ que permite estender sua sintaxe concreta, foi apresentada neste capítulo. O conceito de classes sintáticas foi definido, e discutiu-se como ela é usada para especificar novas extensões. A modularidade das extensões é alcançada com as classes sintáticas porque todo código referente às extensões, sintaxe, semântica e representação na árvore de sintaxe abstrata, é encapsulado em uma única unidade sintática. Portabilidade também é uma vantagem alcançada com o uso da linguagem XAJ, pois as extensões são definidas completamente na própria linguagem, não dependendo de nenhuma ferramenta específica para implementá-las. Também foi apresentada uma especificação formal da gramática de XAJ e como ela foi inserida na gramática da linguagem AspectJ.

As classes sintáticas permitem adicionar quaisquer produções em formato $A \rightarrow A_1 A_2 \ldots A_n$ à gramática da linguagem. Assim, teoricamente, quaisquer extensões ou linguagens de domínio específico que possam ser expressas por gramáticas livre de contexto podem ser embarcadas à linguagem XAJ. Entretanto, algumas extensões podem gerar conflitos, pois a versão do compilador da linguagem usa um algoritmo LALR para gerar a tabela de *parse*.

XAJ permite usar vários passos para transformar a AST, assim é possível coletar informações, que não podem ser capturadas pelo formalismo de gramática livre de contexto, antes de atribuir a semântica da extensão em termos da sintaxe da linguagem base.

Capítulo 4

Implementação da Linguagem

O processo de compilação de um código escrito na linguagem XAJ é dividido em duas etapas. A primeira compila as classes sintáticas, extrai informações para a análise sintática das extensões definidas em cada classe sintática e compila o método *desugar*. Na segunda etapa, o código que usa as extensões é compilado. Quando o compilador encontrar no código fonte uma declaração de importação de sintaxe, ele deve estender a tabela de *parser* da linguagem com as informações para a análise sintática da extensão, que foi obtida no processo de compilação da classe sintática que define a extensão. Depois que a importação de sintaxe não estiver mais no escopo, as informações para realizar a análise sintática da extensão definida por esta importação devem ser retiradas da tabela de *parser*. Portanto, qualquer compilador para a linguagem XAJ deve ser capaz de modificar sua tabela de *parser* em tempo de execução para realizar a análise sintática das extensões definidas pelo "engenheiro de linguagens". Por esse motivo o compilador de XAJ deve ter funcionalidades incomuns.

A Figura 4.1 ilustra a primeira etapa de compilação descrita acima, onde uma classe sintática *Foo* é compilada. Nesse processo são gerados como produtos, informações para a análise sintática da extensão *Foo* e o *bytecode* referente à classe sintática compilada, o qual contém o método *desugar* compilado. A Figura 4.2 mostra o processo onde a tabela de parser do compilador estendida com as informações sintáticas da extensão *Foo* quando um arquivo que importa essa extensão é compilado.

As classes sintáticas representam as extensões na árvore de sintaxe abstrata e o método *desugar* dessas classes é executado em tempo de compilação para implementar a semântica da extensão. Por esses motivos, as classes sintáticas devem ser compiladas antes da compilação das classes que utilizam as extensões definidas por elas. Observe que essa abordagem restrige a utilização da extensão definida por uma classe sintática dentro dela, pois a classe sintática já teria que estar compilada para compilá-la,

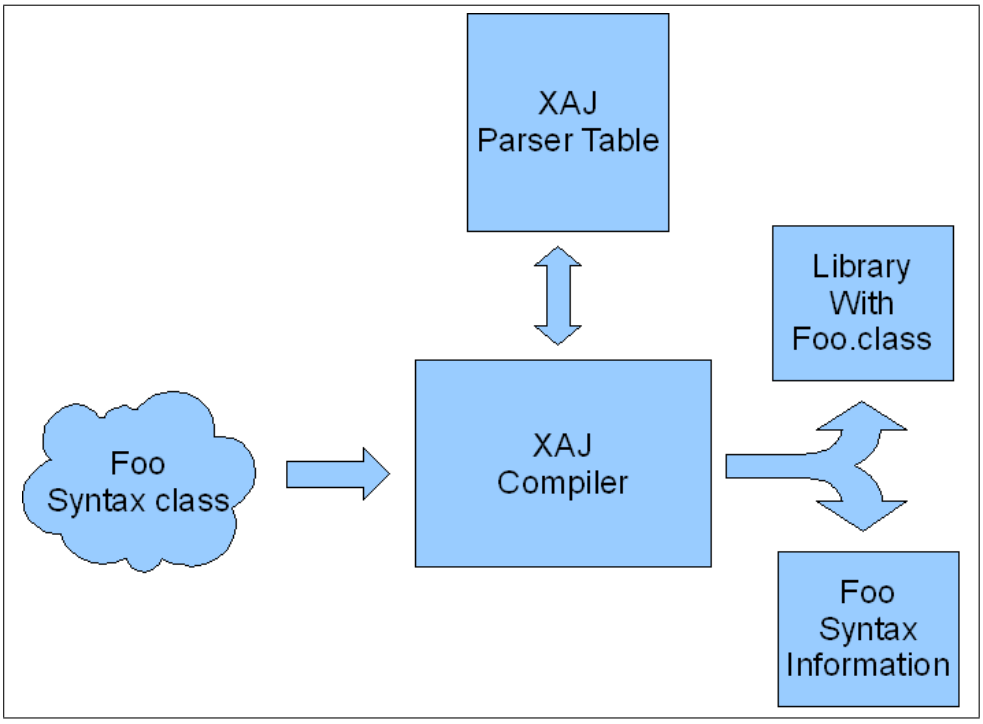

Figura 4.1. Etapa 1: Informações para a análise sintática gerados pelo compilador da linguagem XAJ.

ocorrendo um impasse no compilador de XAJ. Esse problema pode ocorrer envolvendo várias classes sintáticas, quando dependem entre si para serem compiladas, portanto optamos por restringir no compilador de XAJ o uso das novas sintaxes na definição das classes sintáticas.

Um compilador desenvolvido para a linguagem XAJ é descrito neste capítulo. A Seção 4.1 descreve as ferramentas usadas na implementação do compilador, as Seções 4.2 e 4.3 descrevem a arquitetura e os detalhes da implementação, respectivamente. A Seção 4.4 discute os tipos de erros de podem ocorrer e o tratamento dos mesmos e as conclusões são discutidas na Seção 4.5.

4.1 Ferramentas Usadas na Implementação

O AspectBench Compiler (*abc*) [Avgustinov et al., 2005] foi usado para implementar a primeira versão do compilador da linguagem XAJ. O *abc* foi escolhido porque tem uma implementação modular e eficiente para a linguagem AspectJ e foi desenvolvido para ser fácil de estender a fim de testar novas características para a linguagem AspectJ.

Os compiladores *ajc* e um *front-end* para a linguagem AspectJ [Bravenboer et al., 2006b] também foram analisados e poderiam ser usados na implementação do compila-

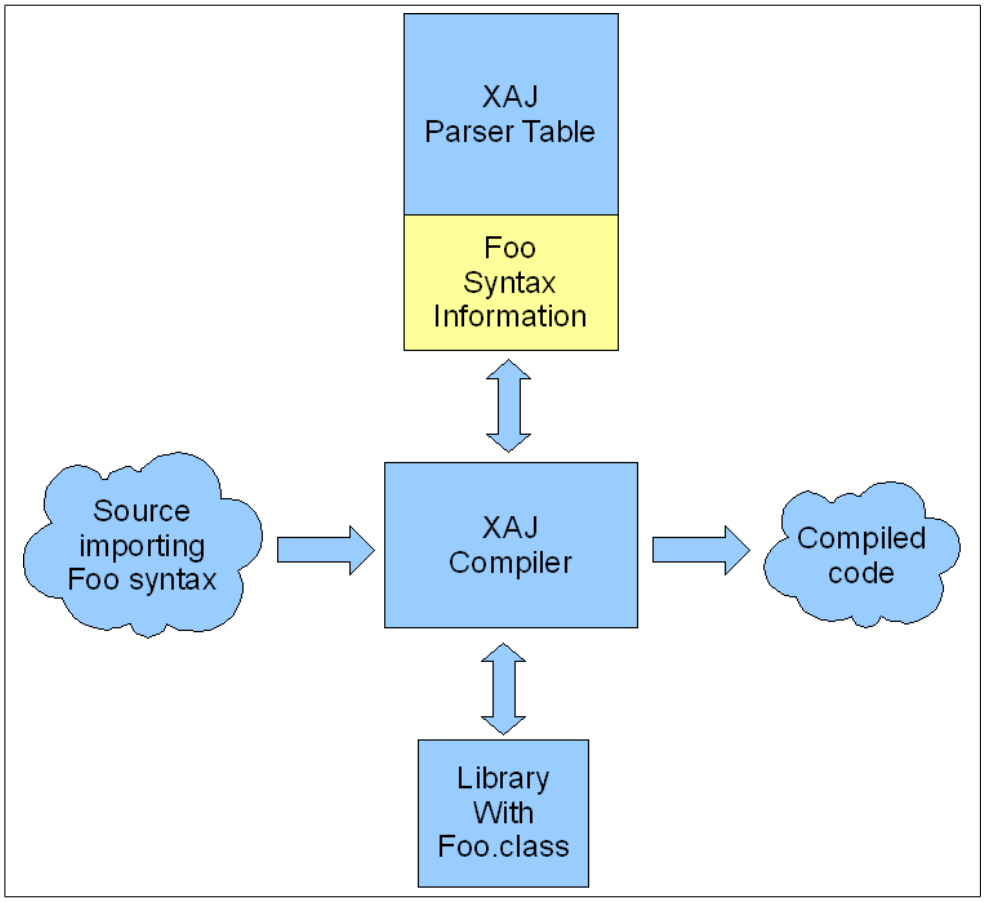

Figura 4.2. Etapa 2: A tabela de parser da linguagem XAJ é estendida para compilar a extensão Foo.

dor para XAJ. *ajc* foi o primeiro compilador para a linguagem AspectJ, originalmente desenvolvido pelos inventores da linguagem e atualmente é mantido como parte do projeto Eclipse AspectJ [AspectJ, b], porém ele foi projetado para realizar uma compilação incremental e eficiente, e não para ser fácil de estender, por isso não foi usado na implementação do compilador de XAJ.

Em 2006 [Bravenboer et al., 2006b], um *front-end* para a linguagem AspectJ foi implementado usando o SDF2 [Visser, 1997b] e apresenta uma definição sintática da linguagem modular e fácil de estender. No entanto, não existe integração com nenhum *back-end* da linguagem AspectJ e, por este motivo, não foi usado para implementar o compilador de XAJ. O restante desta seção irá explicar os mecanismos de extensão usados pelo *abc*.

4.1.1 Polyglot

Polyglot [Nystrom et al., 2003] é um *front-end* extensível para a linguagem Java que contém toda a análise semântica necessária para a linguagem. Polyglot é estruturado como uma sequência de passos, no qual cada passo recebe uma árvore de sintaxe abstrata e realiza operações na AST e tem como saída uma AST modificada. Estruturas auxiliares também são construídas pelo Polyglot, tais como a tabela de símbolos e o sistema de tipos.

Extensões na gramática da linguagem são feitas usando o *PPG* [Brukman & Myers, 2008], um pré-processador para o gerador de analisadores sintáticos LALR CUP [Hudson et al., 1996], que permite definir a gramática da extensão por meio de um conjunto de modificações na gramática original da linguagem. Polyglot faz forte uso de interfaces e fábricas para criar os objetos da árvore de sintaxe abstrata, facilitando estender, modificar e adicionar novos nós ao compilador. No entanto, Polyglot não oferece nenhuma facilidade para estender o analisador léxico da linguagem.

Na programação orientada a objetos, herança é o mecanismo usado para adicionar novos campos, métodos ou sobrescrever o comportamento de alguns métodos, porém quando essas mudanças afetam diversas classes, há uma repetição de código e este processo pode ser tedioso [Nystrom et al., 2003]. Polyglot soluciona esse problema adicionando um objeto de extensão para cada nó da AST que serve para adicionar novos campos ou métodos nas classes. Classes que possuem as mesmas implementações desses campos e métodos têm o mesmo objeto de extensão, evitando assim duplicação de código. A sobrecarga de métodos é feita usando um objeto de delegação do nó, podendo ser ele mesmo.

A maioria dos passos do Polyglot é baseado em uma versão modificada do padrão Visitor, que visita cada nó e delega a operação a ser executada para o nó que está sendo visitado, chamando a função adequada do nó para realizar a operação. Polyglot permite modificar, acrescentar, reordenar ou excluir qualquer passo executado na AST.

Polyglot modifica a AST do programa em cada passo executado substituindo um nó por outro alterado, porém ele não está preparado para substituir um nó da árvore por uma lista de nós. Na linguagem XAJ, em alguns casos, o método *desugar* deverá retornar uma lista de nós, como no exemplo da Figura 1.4. Neste trabalho, foi usada uma versão modificada do Polyglot que suporta uma lista de nós como retorno. Essa modificação do código do Polyglot foi conduzida pelo próprio autor desta dissertação.

4.1.2 AspectBench Compiler

O *front-end* do *abc* foi construído usando Polyglot. Assim, extensões para a linguagem AspectJ que afetam o *front-end* do *abc* são feitas usando os mesmos mecanismos de extensão do Polyglot.

Usando o Polyglot, o *abc* faz a análise sintática do código e ao fim do *front-end* a árvore de sintaxe abstrata é dividida em uma AST com código somente em Java e outra AST com informações de aspectos. Depois dessa divisão, é realizado o processo de costura, no qual a AST em puro Java é modificada de acordo com as informações de aspectos e ao fim deste processo o bytecode é gerado.

As extensões permitidas pelo compilador da linguagem implementado neste trabalho não lidam com extensões em tempo de costura, como apresentado na proposta da linguagem de Di Iorio et al. [2009]. Assim, o compilador para a linguagem XAJ foi implementado estendendo somente o *front-end* do *abc*, antes da AST ser dividida em informações de aspectos e AST em puro Java.

4.2 Arquitetura do Compilador

O *abc* não possui uma tabela de *parser* que possa ser modificada em tempo de execução. Assim, por questões de eficiência, nesta versão do compilador para a linguagem XAJ uma tabela de *parser* incluindo a sintaxe de todas as extensões foi gerada, evitando que uma tabela de *parser* fosse construída diversas vezes. Portanto, todas as classes sintáticas são compiladas separadamente e as informações para o *parser* das extensões são extraídas e depois constrói-se a tabela de *parser* que contém as extensões definidas.

A Figura 4.3 apresenta a arquitetura do compilador desenvolvido, denominado de *xajc (XAJ Compiler)*. O compilador *xajc* separa as classes sintáticas do código, cujas informações de sintaxe das extensões são extraídas e as classes sintáticas compiladas pelo *scc (Syntax Class Compiler)*. O *scc* gera um arquivo com as informações da sintaxe das extensões, que depois é compilada e gera a tabela de *parser* estendida pelo PPG, e os *bytecodes* das classes sintáticas com o método *desugar* compilado. A tabela de *parser* estendida e as classes sintáticas compiladas geradas neste processo são usados pelo *xc (eXtended Compiler)* para compilar o restante do código que usa as extensões. O método *desugar* das classes sintáticas é compilado pelo *scc* e é executado pelo *xc*.

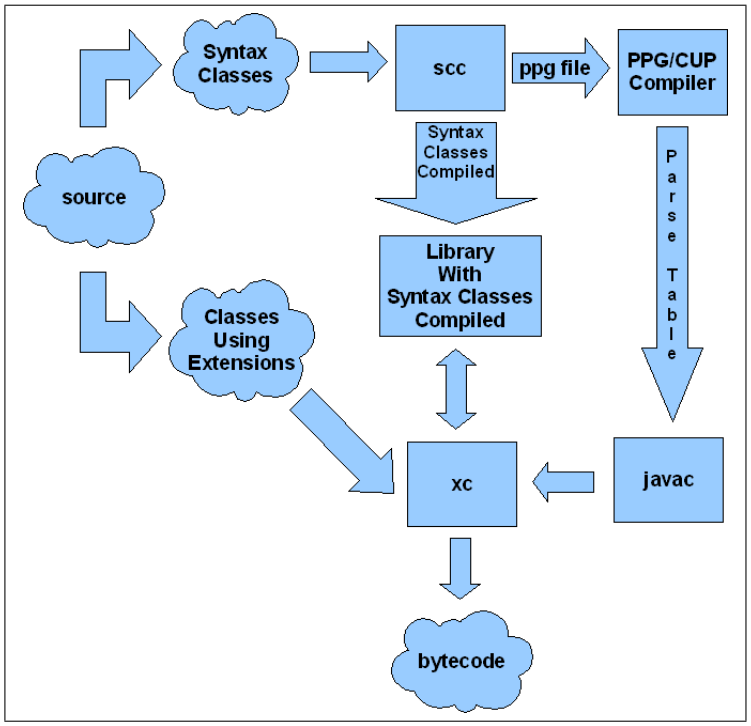

Figura 4.3. Arquitetura do Compilador da Linguagem XAJ.

4.3 Detalhes da Implementação

Nesta seção, será abordada com mais detalhes a implementação de cada um dos componentes do compilador *xajc* descritos anteriormente, o *scc* e *xc*.

4.3.1 Syntax Class Compiler

O *scc* é uma extensão do compilador do *abc* para compilar as classes sintáticas. Ele é um compilador completo para a linguagem AspectJ acrescido das classes sintáticas. A construção deste compilador foi feita estendendo a gramática da linguagem para permitir que as classes sintáticas sejam declaradas (a gramática desenvolvida encontra-se no apêndice A) e novas classes criadas para representar as classe sintáticas na AST. Novos passos foram adicionados ao *front-end* para realizar as tarefas de coleta das informações das sintaxes das extensões, construir o arquivo de entrada para o PPG que irá gerar a tabela de *parser* estendida, coletar o números de passos e geração automática de código. Todos esses passos são realizados depois que a AST para as classes sintáticas é construída. Depois de realizadas estas operações, o compilador *scc* se comporta como o compilador do *abc* e gera os *bytecodes* das classes sintáticas, contendo o método *desugar* compilado, que é adicionado à biblioteca da linguagem para ser usado no *xc* para construir a AST do restante do código. A Figura 4.4 mostra

```
public class MultiIntro extends Term_c implements IntertypeDecl, NodeXAJ {

  public Node desugar(Context ctx, NodeFactory nf,
    TypeSystem ts) {
    ClassDecl cd = ctx.getClass(className);
    ClassDecl sub[] = cd.getSubClasses();
    List list = nf.TypedList(ClassMember.class);
    for(ClassDecl x :  sub)
      list.add(nf.IntertypeMethodDecl(modifiers,
        returntype, x, methodName, params, block));
     return list;
  }
  public MultiIntro(Flags modifiers, ...)  { ...}
  private Flags modifiers;
  public Flags getmodifiers() { return modifiers; }
  public void setmodifiers(Flags modifiers) {
    this.modifiers = modifiers;
  }
  ...
}
```

Figura 4.4. Exemplo da classe sintática MultiIntro decompilada

o código descompilado da classe sintática apresentada na Figura 3.1. Note que o código da classe sintática não contém mais as informações de extensões, sendo traduzida para uma classe comum da linguagem Java com o método *desugar*.

Para gerar o arquivo de entrada para o PPG, contendo as informações da sintaxe das extensões, é necessário transformar as produções definidas nas classes sintáticas para o formato esperado pelo PPG. A maioria das produções têm um mapeamento direto, somente quando os operadores de lista ({ e }) e opcional ([e]) são usados, é que produções auxiliares são geradas. O mapeamento das cadeias de caracteres para os *tokens* correspondentes é feito usando o próprio analisador léxico da linguagem, porém essa abordagem tem como desvantagem os espaços que poderão aparecer entre os símbolos gerados quando é feita análise léxica da extensão. A Figura 4.5 mostra como ficará o mapeamento da extensão *MultiIntro*, definida na Figura 1.4, para o arquivo usado pelo PPG. Note que a cadeia de caracteres "`+.`" foi traduzida para *PLUS DOT*, podendo ter o efeito indesejável de conter espaços entre os símbolo "`+`" e "`.`". Para evitar esse problema, deveria ser aplicada uma abordagem que use um analisador sintático sem o analisador léxico, como o apresentado por Visser [1997a], ou o desenvolvimento de um analisador léxico extensível.

```
MultiIntro ::=
  modifiers_opt:modifiers type:returnType
  identifier:className PLUS DOT
  identifier:methodName LPAREN
  formal_parameter_list_opt:params RPAREN
  block:introducedCode;
```

Figura 4.5. Mapeamento de literal string para tokens do ppg.

4.3.2 Extended Compiler

O arquivo que contém as informações de extensão gerado pelo *scc* é compilado pelo PPG, que gera a tabela de *parser*, que depois é compilada com o compilador *javac*. Esta tabela de *parser* é carregada pelo *xc*, por reflexão, para compilar o código que usa as extensões.

O compilador *xc* também é uma extensão completa do compilador *abc*, assim como o *scc*. No entanto, este compilador é "construído" em tempo de compilação. O *abc* tem uma classe, denominada *ExtensionInfo*, que define os passos do compilador, cria os objetos que realizam a análise léxica e sintática, dentre outras funções. A quantidade de passos que as classes sintáticas necessitam para transformar os nós da extensão em subárvores com nós somente da linguagem AspectJ e a tabela de *parser* só são obtidas depois que o *scc* compila as classes sintáticas, não sendo possível determinar a priori. Para construir o *xc*, a classe *ExtensionInfo* com as informações obtidas ao fim do *scc* é gerada, compilada e carregada, assim como a tabela de *parser*.

Com a tabela de *parser* já carregada, o *xc* constroi a AST do programa, que contém os nós de extensão representadas pelas classes sintáticas. Note que neste ponto, as classes sintáticas já foram compiladas pelo *scc* e adicionadas à biblioteca da linguagem, podendo ser instanciadas pela tabela de *parser*. Depois que a AST foi construída, uma sequência de passos que chama o método *desugar*, com o parâmetro do número de passos corrente de cada classe sintática (neste ponto as classes sintáticas são nós da AST) é executado, e substitui o nó da extensão pela subárvore retornada pelo método *desugar*. Ao término deste processo, a árvore de sintaxe abstrata do programa conterá somente nós da linguagem AspectJ e todos os passos seguintes do compilador do *abc* são executados para compilar a AST modificada e gerar o *bytecode* do programa.

O compilador do *abc* não tem os recursos de extensibilidade da tabela de *parser* em tempo de execução, assim a importação de sintaxe da linguagem foi implementada tendo um efeito global, e não como descrita no Capítulo 3, cujo comportamento da importação de sintaxe altera a gramática localmente. Em versões futuras do compilador,

será adicionada a importação de sintaxe com efeito local.

4.4 Tratamento de Erros

Existem três pontos que podem gerar erros no compilador *xajc*, os erros gerados pelo *scc* referentes às classes sintáticas, os erros gerados pelo *xc* e erros no processo da criação da tabela de *parser*. Nesta versão do compilador, não foi feito nenhum tratamento especial de erros, além dos recursos oferecidos pelo *abc* para isso.

Durante o *scc*, podem ocorrer erros de sintaxe nas classes sintáticas quando é definida a declaração sintática, ou erros da linguagem AspectJ. Esses erros são capturados pelo compilador e é emitida uma mensagem informando a linha e coluna na qual aconteceu. Quando os nomes de não-terminais e terminais são referenciados nas declarações sintáticas, erros também podem ser gerados, como nomes que não pertecem à gramática da linguagem ou que não foram definidos. Esses tipos de erros também são verificados e é informada a linha e coluna, com uma mensagem apropriada.

A tabela de parser é gerada usando um gerador automático LALR, portanto as extensões definidas podem gerar conflitos com outras produções existentes da gramática. Nesta versão, o compilador *xajc* não trata os conflitos gerados, e a mensagem de erro notificada é a mesma emitida pelo gerador LALR, deixando para o "engenheiro de linguagens" a resolução destes problemas. Nas próximas versões, uma mensagem de erro mais apropriada para este tipo de erro deve ser incluída, assim como facilidades para resolver os conflitos.

Quando são compiladas as classes que usam as extensões, pelo *xc*, erros de sintaxe do código AspectJ e do uso das extensões podem acontecer. Uma mensagem de erro informando a linha e coluna onde o erro aconteceu é gerada. Porém, podem ocorrer erros de semântica depois que o método *desugar* substituir o nó da extensão pela subárvore que contém somente construções AspectJ. Quando esse erro ocorrer, é gerada uma mensagem de erro padrão com a linha e coluna, porém a linha e coluna podem não ser as corretas, pois podem ser alteradas no método *desugar* quando a nova AST é construida.

4.5 Conclusão

Neste capítulo, as características gerais de um compilador para a linguagem XAJ e as dificuldades de implementação foram discutidas. Uma primeira versão do compilador para a linguagem XAJ, denominada *xajc*, também foi apresentada.

O compilador do *abc*, ferramenta utilizada no desenvolvimento do *xajc*, não está preparado para que a sua tabela de *parser* seja modificada em tempo de execução, pois isso não é uma característica comum na maioria dos compiladores. Assim, o compilador *xajc* gera uma tabela de *parser* para cada programa que é compilado. No entanto, extensões para a linguagem não afetam significamente a tabela de *parser* e um sistema que permita que a tabela de *parser* da linguagem AspectJ seja modificada para acrescentar essas pequenas modificações aumentaria a eficiência do compilador. Trabalhos como o de Korenjak [1969] podem ser uma solução para essa nova característica do compilador.

O *abc* não oferece recursos para estender o analisador léxico e com isso alguns problemas, como espaçamento indesejado entre *tokens*, podem ser causados pela abordagem usada pelo *xajc*. Este problema poderia ser evitado com o uso de um analisador léxico estensível ou usando um analisador sintático sem análise léxica, e.g., o desenvolvido por Visser [1997a]. Durante a condução deste trabalho, uma nova versão do *front-end* do compilador *abc* foi implementada usando o JastAdd [Ekman & Hedin, 2007], um compilador extensível para Java, obtendo uma implementação menor, mais rápida e mais fácil de estender [Avgustinov et al., 2008]. Esse *front-end* poderá ser usada em versões futuras do compilador.

Eventualmente, as extensões criadas pelos "engenheiros de linguagens" podem gerar conflitos com as construções da linguagem AspectJ ou com outras extensões. Extensões conflitantes que não são usadas no mesmo escopo não deveriam ocasionar erros de conflitos, porém devido à abordagem atual isso pode acontecer, mas com o uso de uma tabela de *parser* extensível este problema deve ser resolvido. Facilidades para solucionar conflitos e ambiguidades possivelmente causadas por novas extensões serão inseridas em versões futuras do compilador *xajc*.

Capítulo 5

Avaliação das Construções de XAJ

Neste capítulo, algumas extensões são implementadas na linguagem XAJ. Na Seção 5.1, uma versão mais complexa da extensão *MultiIntro* é descrita, mostrando as funcionalidades oferecidas pela linguagem XAJ para modularização das extensões. Uma implementação de uma linguagem de domínio específico orientada a aspectos é apresentada na Seção 5.2, e um exemplo que necessita do uso de mais de um passo é apresentado na Seção 5.3.

5.1 MultiIntro

Na Seção 1.1, foi apresentada uma versão da extensão proposta por Chiba & Nakagawa [2004] para resolver o problema de repetição de código para inserir o método *accept* nas classes a serem visitadas pelo padrão *visitor*. Na solução apresentada, o método *accept* de cada subclasse invoca um método de nome *visit*, que difere entre si apenas no parâmetro *this*. Essa solução impõe que o método chamado por cada subclasse para realizar a visitação deva ter o mesmo nome. Porém, o que fazer se o usuário desejar que o nome do método chamado seja diferente para cada subclasse, e.g., *visit* concatenada ao nome da subclasse?

Nesta seção é apresentada uma implementação da extensão *MultiIntro* que tem essa característica. É definida uma extensão que forma um identificador a partir de constantes literais e variáveis que são passadas em tempo de compilação (Seção 5.1.1), depois extensões que permitem chamadas de métodos usando a extensão definida na Seção 5.1.1 (Seção 5.1.2) e blocos no qual a nova maneira de invocar métodos podem ser usadas (Seção 5.1.3) são definidas e por fim, a extensão *MultIntro* é definida usando o novo bloco (Seção 5.1.4).

```
package multiIntro;

import java.util.*;

public syntaxclass Eval {
  @grammar {
    Eval → ‘‘<%’’ l = {id = IDENTIFIER | s = STRING_LITERAL} ‘‘%>’’;
  }

  private static String value = ‘‘’’;
  private static Map entries = new HashMap();

  public static String getEvaluatedValue() {
    return value;
  }

  public static void put(String name, String value) {
    entries.put(name, value);
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    List l = getL();
    Iterator it = l.iterator();
    value = ‘‘’’;
    while (it.hasNext()) {
      NodeXAJ prox = l.next();
      if(prox.getS() != null)
        value += ((polyglot.lex.StringLiteral)
          prox.getS()).getValue();
      else
        value += (String)entries.get(((polyglot.lex.Identifier)
         prox.getId()).getIdentifier());
    }
    return this;
  }
}
```

Figura 5.1. Definição da classe sintática Eval.

5.1.1 Definição da Extensão *Eval*

A Figura 5.1 mostra a classe sintática *Eval*, que contém uma expressão formada por uma sequência de identificadores e literais.O método *desugar* calcula o valor da expressão fazendo a concatenação dos elementos. Os valores dos identificadores são atribuídos usando a função *put* da classe sintática. Um exemplo de uso da extensão *Eval* seria

```
<% ‘‘visit’’ className %>
```

```
package multiIntro;

importsymbol multiIntro.Eval;
import java.util.*;

public syntaxclass MethodInvocationEval {
  @grammar extends method_invocation {
    MethodInvocationEval → n = name ".”  e = Eval
      "(" a = argument_list_opt ")";
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    Name name = getN();
    List args = getA();
    Eval eval = getE();
    return ft.Call(position(), name.prefix == null ?
      name.toReceiver() :  name.prefix.toReceiver(),
       ((Eval) eval.desugar(ft, ts)).getEvaluatedValue(),
         args);
  }
}
```

Figura 5.2. Definição da classe sintática MethodInvocationEval.

Se fosse atribuído ao identificador *className* o valor *Number*, o resultado da execução do método *desugar* seria *visitNumber*.

O *abc* não representa identificadores como nó da árvore de sintaxe abstrata, por isso não é possível retornar a expressão avaliada pelo método desugar para ser substituída pelo nó Eval. Um nó para representar identificadores não foi criado porque para fazer esta alteração no compilador do *abc* seria necessário uma demanda de tempo considerável, pois afetaria todos os nó da AST. Portanto, como não foi possível estender um identificador e para obter o efeito desejado na implementação da extensão *MultiIntro* foi necessário estender a invocação de método, como apresentada na Seção 5.1.2.

5.1.2 Definição da Extensão Método com *Eval*

A extensão *Eval* definida na seção anterior é usada para definir uma extensão para chamadas de métodos com *Eval*, sendo definida pelo objeto alvo seguido por ponto e por um *Eval*. Um exemplo de chamada de método com *Eval* seria

```
v."<%" "visit" className "%>"
```

A Figura 5.2 mostra a definição da classe sintática do método com *Eval*. A classe

```
package multiIntro;

importsymbol multiIntro.MethodInvocationEval;
import java.util.*;

public syntaxclass BlockEval {
  @grammar using MethodInvocationEval {
    BlockEval → b = block;
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    Block b = getB();
    Block newblock = ft.Block(position());
    for(Object ob :  b.statements()) {
      polyglot.ast.Stmt stmt = (polyglot.ast.Stmt) ob;
      newblock = newblock.append(stmt.desugar(ct, ft, ts));
    }
    return newblock;
  }
}
```

Figura 5.3. Definição da classe sintática BlockEval.

sintática especifica que a extensão *MethodInvocationEval* pode ser usada nos mesmos pontos em que uma invocação de método é esperada. O método *desugar* retorna uma chamada de método criado usando o valor da expressão avaliada pela classe sintática *Eval* como nome do método a ser invocado.

5.1.3 Definição da Extensão Bloco com *Eval*

Uma definição de uma extensão de um bloco que pode conter invocações de métodos com *Eval* é apresentada na Figura 5.3. Note que a extensão definida não usa a produção *MethodInvocationEval* especificada na declaração *using*, porém a mesma define que essa extensão é alcançável, podendo ser usada dentro de um bloco, pois MethodInvocationEval estende a invocação de método. Seria muito trabalhoso definir um novo bloco que difere do já existente unicamente pela chamada de método, assim a declaração *using* da linguagem XAJ é muito útil para este caso.

5.1.4 Definição da Extensão *MultiIntro*

Depois de definido um bloco que pode conter uma chamada de método com *Eval*, a extensão *MultiIntro* (Figura 5.4) pode ser definida fazendo uma adaptação da classe sintática apresentada na Figura 3.1 para gerar um *BlockEval* no lugar de *block*. O

```
package multiIntro;

importsymbol multiIntro.BlockEval;
import java.util.*;

public syntaxclass MultiIntro {
  @grammar extends intertype_member_declaration {
    MultiIntro ->
     m = modifiers_opt
     t = type
     c = identifier
     "+." methodName = identifier
     "(" p = formal_parameter_list_opt ")"
     b = BlockEval;
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    Flags modifiers = getM();
    TypeNode returnType = getT();
    String methodName = getMethodName();
    List params = getP();
    BlockEval block = getB();
    ClassDecl cd = ctx.getClass(getC());
    ClassDecl sub[] = cd.getSubClasses();
    List list = nf.TypedList(ClassMember.class);
    for(ClassDecl x :  sub)
      BlockEval b = block.copy();
      Eval.put("className", x.getName());
      list.add(nf.IntertypeMethodDecl(modifiers,
        returntype, x, methodName, params,
          (Block) b.desugar(ct, nt, ts)));
     return list;
  }
}
```

Figura 5.4. Definição da classe sintática MultiIntro com Eval.

método *desugar* faz uma cópia do bloco e altera o valor do identificador *className* da classe sintática *Eval* para o nome da classe corrente antes de executar o método *desugar* para o mesmo.

5.2 AJSynchro

Em 2008, uma linguagem de domínio específico orientada a aspectos, denominada AJSynchro, foi desenvolvida por Di Iorio et al. [2008] para realizar a sincronização de um sistema para construção de roteiros em uma malha urbana. A linguagem AJSynchro define uma nova entidade denominada *Synchronizer*, que especifica qual classe Java do sistema deve ser sincronizada. A entidade *Synchronizer* é similar a uma classe Java e pode ter como membros um conjunto de atributos que são inseridos na classe a ser sincronizada e adendos para sincronizar métodos da classe especificada.

Nas próximas seções, a linguagem AJSynchro é implementada em XAJ e a sincronização do exemplo do produtor/consumidor é feita usando a linguagem AJSynchro.

5.2.1 Definição dos Comandos Await e Signal

A linguagem AJSynchro define dois novos comandos, *await* e *signal*, cuja sintaxe são, respectivamente:

```
"await" condIdentifier "while" "(" expression ")" ";"
```

```
"signal" condIdentifier ";"
```

onde *condIdentifier* é um identificador do tipo *Condition* e *expression* é uma expressão booleana válida. A semântica do comando *await* é colocar o objeto para dormir enquanto a expressão especificada não for verdadeira, e o comando *signal* tem como semântica acordar o objeto. As Figuras 5.5 e 5.6 mostram a definição das classes sintáticas que implementam os comandos *signal* e *await*, respectivamente.

5.2.2 Definição da Classe Sintática AJSynchronize

AJSynchro tem uma espécie de adendo de sincronização (*synchronize*), que especifica um conjunto de métodos que serão sincronizados da classe em questão. O adendo *synchronize* pode conter, opcionalmente, um *lock* e comandos *before* e *after*, que definem

```
package ajsynchro;

import java.util.*;

public syntaxclass Signal {
  @grammar extends statement {
    Signal -> “signal” id = simple_name “;” ;
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    List list = new TypedList(new LinkedList(),
      Expr.class, false);
    Name name = getId();
    Stmt stmt = nf.Eval(position(), nf.Call(position(),
      name.toExpr(), "signal", list));
     return stmt;
  }
}
```

Figura 5.5. Definição da classe sintática Signal.

```
package ajsynchro;

import java.util.*;

public syntaxclass Await {
  @grammar extends statement {
    Await -> “Await” id = simple_name
      “while” “(” e = expression “)” “;”;
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    List list = new TypedList(new LinkedList(),
      Expr.class, false);
    Name name = getId();
    Expr expr = getE();
    Block body = nf.Block(position(), nf.Eval(position(),
      nf.Call(position(), name.toExpr(), "await")));
     return nf.While(position(), expr, body);
  }
}
```

Figura 5.6. Definição da classe sintática Await.

```
package ajsynchro;
importsymbol ajsynchro.Await;
importsymbol ajsynchro.Signal;
import java.util.*;
public syntaxclass AJSynchronize {
  @grammar extends advice_declaration {
    AJSynchronize -> "synchronize" "(" m = method_header
      l = {"," n = method_header} ")"
      ["with" id = simple_name]
      "{" ab = [AJBefore] aa = [AJAfter] "}";
  }
  public static syntaxclass AJBefore {
    @grammar using Signal, Await {
    AJBefore -> "before" ":"  b = block_statements;
    }
    public Node desugar(Context ct,
      NodeFactory ft, TypeSystem ts) {
      List statements = getB();
      return nf.Block(position(), statements);
    }
  }
  public static syntaxclass AJAfter {
    @grammar using Signal, Await {
    AJAfter -> "after" ":"  b = block_statements;
    }
    /* Desugar similar a AJBefore */
  }
  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
    ListNode l = new ft.newListNode();
    for(/*cada método sincronizado*/) {
      /* Cria um ponto de junção do tipo execution
      * que captura o padrão do método sincronizado */
      /* Cria um bloco try-catch a partir de lock e
      * os comandos before e after, se existirem */
      /*Adiciona o adendo around criado a partir
      * do ponto de junção e dos blocos criados
      * anteriormente à l
    }
    return l;
  }
}
```

Figura 5.7. Definição da classe sintática AJSynchronize.

quais comandos serão executados antes e depois da execução dos métodos sincronizados, respectivamente.

A Figura 5.7 apresenta a classe sintática que implementa a extensão do adendo *synchronize*. O método *desugar* desta classe sintática retorna uma lista de adendos do tipo *around* cujo ponto de junção captura a execução de cada método especificado. O

```
package ajsynchro;
importsymbol ajsynchro.AJSynchronize;
import java.util.*;
public syntaxclass AJSynchronizer {
  @grammar extends aspect_declaration {
    AJSynchronizer -> "synchronizer"
      t = name b = AJSynchroBody;
  }
  public static syntaxclass AJSynchroBody {
    @grammar {
    AJSynchroBody -> "{" {AJSynchroBodyMembers} "}";
    AJSynchroBodyMember -> m = class_member_declaration
      | asm = AJSynchronize;
    }
    ...
  }
  ...
}
```

Figura 5.8. Definição da classe sintática AJSynchronizer.

corpo do adendo é definido a partir dos comandos dos blocos *before*, *after* e do *lock*. A classe sintática *AJSynchronize* contém duas classes sintáticas, *AJBefore* e *AJAfter*, que definem as extensões dos comandos *before* e *after*, respectivamente. O método *desugar* das extensões *AJBefore* e *AJAfter* retornam um bloco de comandos.

5.2.3 Definição da Classe Sintática AJSyncrhonizer

A classe sintática que define a entidade *AJSynchronizer* é apresentada na Figura 5.8. A extensão *AJSynchronizer* contém o nome da classe a ser sincronizada e um corpo, que é definido pela classe sintática interna *AJSynchroBody*. O método *desugar* da extensão *AJSynchronizer* retorna um aspecto cujo corpo é o objeto retornado pelo método *desugar* da classe sintática *AJSynchroBody*.

O método *desugar* da classe sintática *AJSynchroBody* cria uma declaração intertipo para todos os membros que são atributos e retorna um corpo de um aspecto formado pelas declarações intertipos, adendos *around* resultantes da execução do método *desugar* da classe sintática *AJSynchronize* e demais membros de classes.

5.2.4 Produtor-Consumidor Sincronizado com AJSynchro

Para ilustrar como funciona a linguagem AJSynchro, foi feita a sincronização do exemplo do Produtor-Consumidor. O comportamento do Produtor-Consumidor implementado é dado da seguinte forma: O Produtor de posse do processador, apenas escreve

```
public class BufferSimples implements Buffer {
  private int value;
  private static Buffer instance = new BufferSimples();

  private BufferSimples(){
    this.value = -1;
  }

  public static Buffer getInstance() {
    return instance;
  }

  public int getBuffer(){
    System.out.println("Consumidor obter valor = "+ value);
    return value;
  }

  public void setBuffer(int a){
    System.out.println("Produtor produz valor = "+ a);
    this.valor = a;
  }
}
```

Figura 5.9. Classe Buffer do problema Produtor-Consumidor

um inteiro no objeto da classe Buffer, sua escrita ocorre no método *setBuffer*. Analogamente, o Consumidor, de posse do processador obtém o valor corrente que está no Buffer pelo método *getBuffer*. Com essa configuração, é possível ter resultados indesejados, e.g., o Produtor escreve um valor inteiro dentro do Buffer, ainda de posse do processador, o Produtor acaba escrevendo novamente um valor no Buffer sobrescrevendo o antigo valor antes do Consumidor obtê-lo. Essa é apenas uma das situações que resultam em saídas indesejadas, a solução para esse caso é conseguido por meio da sincronização do objeto Buffer.

A Figura 5.9 mostra como foi implementada a classe *Buffer* do exemplo Produtor-Consumidor. O *Buffer* implementado é composto de apenas uma célula , representado por um valor inteiro. Esta classe não possui código relativo a sincronização, sendo que os métodos *getBuffer* e *setBuffer* são bastante coesos, i.e., executam apenas suas responsabilidades de leitura e escrita.

A sincronização do Buffer foi feita utilizando a linguagem AJSynchro, como apresentado na Figura 5.10. A entidade *synchronizer* da Figura 5.10 especifica que a classe *BufferSimples* deve ser sincronizada. Para tal finalidade são inseridos na classe novos atributos do tipo *Condition* e *Lock* para colocar os objetos para dormir quando necessário e um atributo booleano cujo valor é verdadeiro se é possível ler o

```
importsyntax ajsynchro.AJSynchronizer;
synchronizer Buffer {
  Lock accessLock = new ReentrantLock();
  Condition podeEscrever = accessLock.newCondition();
  Condition podeLer = accesslock.newCondition();
  boolean ocupado = false;

  synchronize(public void setBuffer(int)) with accessLock {
    before :
      await podeEscrever while(ocupado);
    after :
      ocupado = true;
      signal podeLer;
  }

  synchronize(public int getBuffer()) with accessLock {
    before :
      await podeLer while(!ocupado);
    after :
      ocupado = false;
      signal podeEscrever;
  }
}
```

Figura 5.10. BufferSimples Sincronizado com a Linguagem AJSynchro

valor do *Buffer*. Os métodos *getBuffer* e *setBuffer* são sincronizados usando o adendo *synchronize*, ambos com o *Lock accessLock*. Os comandos que são executados antes e depois da chamada de cada um dos métodos *getBuffer* e *setBuffer* são semelhantes, os quais colocam o objeto para dormir enquanto não podem ler (ou escrever) antes de executar o método e depois altera o valor da variável *ocupado* e acorda os objetos que estavam dormindo.

Esse exemplo do produtor-consumidor é bem simples, e serve apenas para mostrar o funcionamento da DSAL embarcada, pois a sincronização desse exemplo poderia ser simulado com herança simples, já que a sincronização é aplicada a um método específico de cada vez. As construções da linguagem AJSynchro são bem distantes dos padrões difíceis de entender da linguagem AspectJ, restrige a sincronização aos corpos de métodos e permite aplicar um mesmo tipo de sincronização a diversos métodos que precisam de sinconização equivalente. Segundo Di Iorio et al. [2008], a linguagem foi usada com sucesso na sincronização de um sistema para cálculo de rotas urbanas, evidenciando o desenvolvimento "orientado a uma aplicação" pregado pela metodologia de programação orientada a linguagens, no qual a complicada sincronização do sistema de geração de rotas urbanas foi expressa com uma linguagem relativamente simples e restrita.

5.3 Global Pointcut

Avgustinov [Aske et al., 2005] propôs uma extensão para a linguagem AspectJ, denominada *Global Pointcut*, para eliminar a repetição de código comum a vários adendos, os quais têm uma parte do seu ponto de junção comum a diversos outros. A idéia da extensão *Global Pointcut* é escrever esse código compartilhado uma única vez. Um exemplo de uso seria restringir que adendos sejam aplicados em um conjunto de classes, e.g.,

```
global: * : !within(Hidden);
```

A sintaxe da extensão *Global Pointcut*, proposto em [Aske et al., 2005], é

```
global: <ClassPattern> : <Pointcut>;
```

cuja semântica é adicionar o ponto de junção especificado em todos os pontos de junção dos aspectos definidos por *ClassPattern*. No exemplo apresentado, são excluído todos os conjuntos de junção que ocorrem dentro da classe *Hidden*.

A implementação desta extensão requer dois passos sequenciais. No primeiro, todos os *global pointcut* declarados no programa devem ser adicionados em uma lista. No segundo passo, os pontos de junção de cada adendo devem ser modificados de acordo com os *global pointcut* declarados.

A Figura 5.11 apresenta a implementação da extensão *Global Pointcut* em XAJ. A classe sintática *GlobalPointcut* define a sintaxe da extensão e o método *desugar* coleta as informações do *global pointcut* e as adicionam na lista de *global pointcuts*. A classe sintática *AdviceGP* definida internamente na classe sintática *GlobalPointcut* modifica os nós que são declarações de adendos, por meio do método *desugar* que retorna um novo adendo cujo ponto de junção é modificado com os pontos de junção especificados pelos *global pointcut* do programa. O método *desugar* da classe sintática *AdviceGP* testa se o passo corrente é o segundo, de modo que modifica os adendos somente depois que todos os *global pointcuts* do programa foram coletados.

5.4 Conclusão

Neste capítulo, algumas extensões foram implementadas na linguagem XAJ, cuja finalidade foi apresentar as características da linguagem. Na Seção 5.1, o exemplo apresentado mostra as funcionalidades da linguagem para criar novas extensões a partir de outras existentes e as facilidades oferecidas pelo *using* para criar extensões. O mecanismo do *using* oferecido pela linguagem XAJ permite adicionar novas características à

```
import java.util.*;

public syntaxclass GlobalPointcut {
  @grammar extends class_member_declaration {
    GlobalPointcut -> “global” “:”  c = classname_pattern_expr
     “:”  p = pointcut_expr “;”;
  }
  private static HashMap map = new HashMap();
  public static HashMap getGlobalPointcuts() { ...}

  public static syntaxclass AdviceGP {
    @numberOfPasses = 2;
    @grammar overrides advice_declaration;

    public Node desugar(Context ct, NodeFactory ft,
      TypeSystem ts, int pass) {
      if(pass == 2) {
        for(ClassPattern c :  map.keys())
          if(parent instaceof c)
            /* Cria novo advice que contém o pointcut de map.get(c) */
      }
      return this;
    }
  }

  public Node desugar(Context ct, NodeFactory ft,
    TypeSystem ts) {
   /* Registra todos os pontos em uma mapeamento global*/
    map.add(getC(), getP());
     return this;
  }
}
```

Figura 5.11. Extensão Global Pointcut

linguagem em contextos bem determinados, modificando localmente as regras da gramática. A definição do bloco com invocação de método com *Eval* usa essa característica do *using* (Seção 5.1.3).

Devido à limitação da ferramenta utilizada para implementar a primeira versão do compilador da linguagem XAJ, que não representa identificadores como nós da AST, o método *desugar* da extensão *Eval* definida na Seção 5.1.1 não pode retornar uma nova AST, e apenas calcula o valor da expressão e o armazena em uma variável para ser obtido posteriormente.

Uma linguagem de domínio específico embarcada pode ser vista como um conjunto de extensões para a linguagem hospedeira. Deste modo, o exemplo apresentado na Seção 5.2 mostra a implementação da linguagem de domínio específico orientada a aspectos AJSynchro, a partir de definições de extensões que compõem a linguagem. Di

Iorio et al. [2008] usaram SDF para especificar a sintaxe da linguagem AJSynchro, que permite fazê-la de maneira modular. A sintaxe da linguagem AJSynchro definida em XAJ também foi feita de maneira modular, via classes sintáticas. No entanto, o código referente à semâtica da linguagem é extenso e tedioso de escrever, mas espera-se que a utilização de programação generativa usando o mecanismo de *quasi-quote* facilite a definição da semântica. A semântica da linguagem AJSynchro, no artigo de Di Iorio et al. [2008], foi definida usando as ferramentas de transformação Stratego/XT, sendo trabalhosa também, além de ser uma outra linguagem, ao contrário de XAJ, no qual a semântica é especificada na mesma linguagem que o programador está habituado a programar.

Na Seção 5.3, o exemplo do *Global Pointcut* mostra a utilidade de mais de um passo na execução do método *desugar* e do uso de *overrides*. Em comparação com a implementação para esta extensão dada por Aske et al. [2005], a quantidade de código escrito na linguagem XAJ é menor, pois só é necessário definir a sintaxe e semântica na classe sintática da extensão, ao passo que para implementá-la no *abc*, deve-se alterar o analisador léxico, definir um novo arquivo que especifica a sintaxe da extensão, criar novas classes e interfaces, assim como uma nova fábrica, dentre outras tarefas que devem ser feitas para criar um novo compilador para a linguagem estendida.

Capítulo 6

Conclusão

Nos últimos anos, *linguagens de domínio específico* é um tópico que está em evidência devido aos benefícios proporcionados pelo seu uso no desenvolvimento de sistemas, i.e., ganho em manutenibilidade e produtividade. No entanto, o uso de DSLs no sistema transfere o custo de manutenção para o desenvolvimento e implementação das DSLs que, na maioria das vezes, é realizada pelos desenvolvedores do sistema em que as DSLs serão usadas. Linguagens Extensíveis, cujo interesse da comunidade científica e industrial tem crescido, pode ser uma alternativa para implementação de linguagens de domínio específico, podendo diminuir o custo de manutenção e desenvolvimento das DSLs, eliminando a necessidade de um novo compilador para elas, sendo necessário só estender a linguagem. Linguagens Extensíveis também podem facilitar a integração das novas DSLs com ferramentas de desenvolvimento, além de aumentar a portabilidade das DSLs.

Neste trabalho, a linguagem XAJ, uma linguagem extensível orientada a aspectos, foi definida e uma versão do compilador para a mesma foi desenvolvido. A linguagem XAJ possibilita usar a metodologia de Programação Orientada a Linguagens no desenvolvimento de sistemas junto com o paradigma de Orientação a Aspectos, permitindo, assim, criar linguagens de domínio específico orientada a aspectos embarcadas. XAJ contém característica que permite ao usuário criar e usar extensões e DSALs de maneira modular, por meio classes sintáticas e dos mecanismos de importação de símbolo e sintaxe, e escopo das extensões. Porém, a versão do compilador de XAJ não permite criar extensões que afetem o processo de costura do compilador e nem usar a sintaxe das novas extensões na definição das classes sintáticas.

A linguagem XAJ pode ser comparada com outras abordagens em relação aos mecanismos de extensibilidade da linguagem e com abordagens para embarcar DSALs à linguagem AspectJ. A linguagem Fortress tem um mecanismo de macro para esten-

der a sintaxe concreta da linguagem e, assim com XAJ, Fortress divide o processo de analise sintática em duas etapas. Porém, as macros em Fortress devem ser definidas todas antes da expressão principal e só têm efeito no programa no qual foram definidas, além de não ter um mecanismo de importação da sintaxe. Assim como XAJ, XMF apresenta um mecanismo de importação de sintaxe permitindo que as extensões para a linguagem não afetem globalmente a gramática da linguagem base. As classes sintáticas da linguagem XAJ foram inspiradas nas "syntax class" da linguagem XJ, tendo como vantagem em relação ao mecanismo de expansão de macros a possibilidade de obter informações de contexto antes de fornecer a semântica da extensão, além disso XAJ não usa um caractere de escape antes do uso da extensão como em XMF e XJ. Uma vantagem do uso de XAJ em relação a outras abordagens para embarcar linguagens de domínios específico em AspectJ, tais como MetaBorg, XAspect e *abc*, é que o "engenheiro de linguagens" não necessita aprender diferentes ferramentas e métodos para isto, podendo trabalhar na própria linguagem em que está habituado a programar, além das extensões serem independentes de ferramentas para implementação, pois estão definidas na própria linguagem. As contribuições deste trabalho estão listadas abaixo:

- Desenvolvimento de uma linguagem extensível orientada a aspectos;
- Implementação de um compilador para a linguagem desenvolvida;
- Modularização das extensões via classes sintáticas;
- Portabilidade das DSALs embarcadas na linguagem, pois são definidas na linguagem que é portável e não depende de ferramentas específicas para implementá-las;
- Mecanismo de importação de sintaxe, permitindo que extensões sejam usadas em pontos específicos;
- Mecanismo de importação de símbolos e uso, facilitando a definição de novas extensões baseadas na definição de outras já existentes;
- Definição do *using*, de modo que novas extensões podem ser definidas de maneira sucinta e não afeta globalmente a gramática da linguagem base.

6.1 Trabalhos Futuros

Este trabalho abre uma gama de possibilidades de trabalhos futuros, os quais podem ser divididos entre as duas categorias: *Evolução da linguagem XAJ* e *Ferramentas e*

Ambientes para desenvolvimento e uso de linguagens extensíveis.

Na primeira categoria, Evolução da linguagem XAJ, se enquadram os trabalhos a serem feitos para melhorar a linguagem e a implementação das funcionalidades que não foram implementadas neste trabalho, como extensão que afetam o tempo de costura. Programação generativa usando o mecanismo de quasi-quote e extensões que afetam o processo de costura são características que devem ser adicionadas à linguagem futuramente. A implementação da linguagem não trata do problema de macros higiênicas, assim como fizeram as linguagens Fortress [Allen et al., 2009] e π [Knöll & Mezini, 2009]. Uma versão adaptada do algoritmo usado para a linguagem Lisp e Scheme, descrita em [Clinger & Rees, 1991], deve ser implementada para a linguagem XAJ.

Em alguns casos, o método *desugar* das extensões precisa obter algumas informações de contexto para gerar a árvore de sintaxe abstrata a ser retornada. Na implementação atual, para obter estas informações, é necessário conhecer como o *abc* as representa. A construção de uma biblioteca de manipulação da AST com operações sobre ela é parte dos nossos trabalhos futuros.

As maiores dificuldades encontradas para implementar o compilador da linguagem XAJ estão relacionadas às ferramentas usadas na implementação, pois não há geradores de analisadores sintáticos que tenham facilidades para estender a tabela de *parser*. Além disso, existem poucos compiladores que geram código na linguagem AspectJ. Neste sentido, trabalhos futuros para desenvolver as ferramentas adequadas para a implementação de linguagens extensíveis, como XAJ, devem ser realizados.

A solução encontrada para estender a gramática da linguagem foi gerar um novo analisador sintático em tempo de compilação que contém as produções referentes às novas extensões para a linguagem. As extensões para a linguagem XAJ não afetam significamente a tabela de *parser*, pois as extensões são pontuais e locais, e construir um sistema que permita modificar a tabela de *parser* pontualmente, aumentando a eficiência do processo de compilação da linguagem XAJ, faz parte de projetos futuros de pesquisa.

O mecanismo de importação sintática com efeito local não foi implementado ainda, assim todas as extensões têm efeito global, podendo gerar mais conflitos. O mecanismo de importação sintática com efeito local será implementado futuramente, assim como facilidade para resolver os possíveis conflitos e ambiguidade que podem ocorrer quando uma extensão é definida e a emissão de erros mais apropriados.

O *abc* não possui um analisador léxico extensível, no entanto, algumas extensões necessitam adicionar novos *tokens*, como no Exemplo 5.1 no qual um novo token deveria ser criado para os símbolos "<%" e "%>". A linguagem XAJ prevê a extensibilidade léxica, no entanto, devido às dificuldades em estender o analisador léxico do *abc*,

eles são representados cada um como os *tokens* para "<", "%" e ">" separadamentes. Esta estratégia pode causar efeitos indesejáveis, tais como espaços em branco entre os símbolos e também pode ser fontes de novos conflitos. Por isso, pretende-se realizar a análise sintática sem o analisador léxico, como usado nas ferramentas SDF [Visser, 1997a] e Rats! [Grimm, 2006].

A maioria dos programadores e empresas usam ferramentas de desenvolvimento, e.g., Eclipse e Netbeans, que contêm facilidades que ajudam no desenvolvimento dos programas, como colorir o código-fonte. Uma característica importante de linguagens extensíveis é a interação delas com essas ferramentas de desenvolvimento de modo que possa ser feita a verificação sintática e coloração do código das extensões adicionadas à linguagem, dentre outros utilitários oferecidos pelas ferramentas de desenvolvimento. A interação da linguagem XAJ com ferramentas de desenvolvimento faz parte de nossos trabalhos futuros.

Apêndice A

Gramática da Linguagem XAJ

A gramática da linguagem XAJ é definida a partir da gramática da linguagem AspectJ. A gramática da linguagem na forma que foi utilizada para gerar a tabela de parser é apresentada abaixo. As definições das regras estão no formato esperado para o pré-processador para o gerador de analisadores sintáticos LALR CUP [Hudson et al., 1996], o *PPG* [Brukman & Myers, 2008] que permite estender uma gramática escrita em CUP ou PPG. A principal regra usado na definição da gramática XAJ é a

$$extends\ S ::= <productions>;$$

que adiciona as produções especificadas por $<productions>$ às geradas pelo não terminal S. As gramáticas das linguagens AspectJ e Java que são usadas na definição da gramática de XAJ podem ser encontradas nos apêndices B e C, respectivamente.

Gramática da Linguagem XAJ

```
1  // ------- Productions -----------------------------
2
3  start with goal;
4
5  /* add the possibility of declaring a syntax class to type_declaration */
6  extend type_declaration ::=   // class | interface | aspect | syntaxclass
7       syntaxclass_declaration;
8
9  /* add the possibility of declaring a syntaxclass as class member */
10 extend class_member_declaration ::=
11      syntaxclass_declaration;
12
13 syntaxclass_declaration ::=
14      modifiers_opt SYNTAXCLASS IDENTIFIER super_opt interfaces_opt syntaxclass_body;
15
```

```
16 syntaxclass_body ::=
17      LBRACE numberofpasses_opt syntax_declaration class_body_declarations_opt RBRACE;
18
19 numberofpasses_opt ::=
20      NUMBEROFPASSES EQ INTEGER_LITERAL SEMICOLON
21     |
22       ;
23
24 syntax_declaration ::=
25      GRAMMAR using_opt syntax_declaration_body
26     |
27      GRAMMAR EXTENDS IDENTIFIER using_opt syntax_declaration_body
28     |
29      GRAMMAR OVERRIDES IDENTIFIER SEMICOLON;
30
31 using_opt ::=
32      USING list_identifier
33     |
34       ;
35
36 list_identifier ::=
37      list_identifier COMMA IDENTIFIER
38     |
39       IDENTIFIER;
40
41 syntax_declaration_body ::=
42      LBRACE grammar_productions RBRACE;
43
44 grammar_productions ::=
45      grammar_productions IDENTIFIER ARROW grammar_expression SEMICOLON
46     |
47      IDENTIFIER ARROW grammar_expression SEMICOLON;
48
49 grammar_expression ::=
50      grammar_expression OR production_element
51     |
52      production_element;
53
54 production_element ::=
55         grammar_term semantic_action_opt
56     |
57     /* LAMBDA followed by optional semantic action */
58     semantic_action_opt;
59
```

```
60  grammar_term ::=
61       grammar_term grammar_factor
62      |
63       grammar_factor;
64
65  grammar_factor ::=
66       IDENTIFIER:
67      |
68       IDENTIFIER EQ IDENTIFIER:
69      |
70       STRING_LITERAL
71      |
72       LPAREN grammar_expr RPAREN
73      |
74       named_opt LPAREN grammar_expr RPAREN
75      |
76       LBRACK grammar_expr RBRACK
77      |
78       named_opt LBRACK grammar_expr RBRACK
79      |
80       LBRACE grammar_expr RBRACE
81      |
82       named_opt LBRACE grammar_expr RBRACE;
83
84  named_opt ::=
85       IDENTIFIER EQ;
86
87  grammar_expr ::=
88       grammar_expr OR grammar_term
89      |
90       grammar_term
91       |
92       /* LAMBDA as a grammar_term */
93        ;
94
95  semantic_action_opt ::=
96       LBRACE COLON block_statements_opt COLON RBRACE
97      |
98        ;
99
100 extend import_declaration ::=
101      import_syntax
102     |
103      import_symbol;
```

```
104
105 import_syntax ::=
106      IMPORTSYNTAX qualified_name SEMICOLON
107     |
108      IMPORTSYNTAX name DOT MULT SEMICOLON;
109
110 import_symbol ::=
111      IMPORTSYMBOL qualified_name SEMICOLON
112     |
113      IMPORTSYMBOL name DOT MULT SEMICOLON;
```

Apêndice B

Gramática da Linguagem AspectJ

A gramática da linguagem AspectJ usada neste trabalho é a disponibilizada pela desenvolvedores do *abc*. Esta gramática é a usada para gerar o analisador sintático LALR. A gramática da linguagem AspectJ foi feita estendendo a gramática da linguagem Java. A gramática da linguagem AspectJ é apresentadas abaixo.

Gramática da Linguagem AspectJ

```
1  /* abc - The AspectBench Compiler
2   * Copyright (C) 2004 Laurie Hendren
3   * Copyright (C) 2004 Oege de Moor
4   * Copyright (C) 2004 Aske Simon Christensen
5   *
6   * This compiler is free software; you can redistribute it and/or
7   * modify it under the terms of the GNU Lesser General Public
8   * License as published by the Free Software Foundation; either
9   * version 2.1 of the License, or (at your option) any later version.
10  *
11  * This compiler is distributed in the hope that it will be useful,
12  * but WITHOUT ANY WARRANTY; without even the implied warranty of
13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
14  * Lesser General Public License for more details.
15  *
16  * You should have received a copy of the GNU Lesser General Public
17  * License along with this compiler, in the file LESSER-GPL;
18  * if not, write to the Free Software Foundation, Inc.,
19  * 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
20  */
21
22 /**
23 * @author Laurie Hendren
24 * @author Oege de Moor
```

```
25 * @author Aske Simon Christensen
26 */
27
28 start with goal;
29
30 /* ------------------------------------------------------------------ */
31 /*             EXTENSIONS TO BASE JAVA RULES                          */
32 /* ------------------------------------------------------------------ */
33
34 /* add the possiblility of declaring an apspect to type_declaration */
35 extend type_declaration ::=   // class | interface | aspect
36      aspect_declaration;
37
38 /* add the possibility of a call to proceed as a method_invocation */
39 extend method_invocation ::=
40      PROCEED LPAREN argument_list_opt RPAREN;
41
42 /* add the possiblity of a simple_name for this_ aspectj vars */
43 /*
44 extend simple_name ::=
45      THISJOINPOINT
46   |
47      THISJOINPOINTSTATICPART
48   |
49      THISENCLOSINGJOINPOINTSTATICPART;
50   */
51
52
53 extend class_member_declaration ::=
54     aspect_declaration
55 |
56     pointcut_declaration;
57
58 extend interface_member_declaration ::=
59     aspect_declaration
60 |
61    pointcut_declaration;
62
63 /* ------------------------------------------------------------------ */
64 /*                       ASPECTJ RULES                                */
65 /* ------------------------------------------------------------------ */
66
67 /* must explicitly give two alternatives here,  if you make PRIVILEGED
68    another rule which can go to epsilon,  then there is a shift reduce
```

```
69     conflict with modifiers_opt, which can also go to epsilon. */
70
71  aspect_declaration ::=
72        modifiers_opt PRIVILEGED modifiers_opt ASPECT identifier
73                            super_opt interfaces_opt  perclause_opt
74                            aspect_body
75      |
76        modifiers_opt ASPECT identifier
77                            super_opt interfaces_opt  perclause_opt
78                            aspect_body;
79
80  perclause_opt ::=
81       perclause
82    |
83     /* epsilon */
84        ;
85
86  perclause ::=
87       PERTARGET LPAREN pointcut_expr RPAREN
88    |
89       PERTHIS LPAREN pointcut_expr RPAREN
90    |
91       PERCFLOW LPAREN pointcut_expr RPAREN
92    |
93       PERCFLOWBELOW LPAREN pointcut_expr RPAREN
94    |
95       ISSINGLETON
96    |
97       ISSINGLETON LPAREN RPAREN;
98
99  aspect_body ::=
100      LBRACE  RBRACE
101   |
102      LBRACE aspect_body_declarations RBRACE;
103
104 aspect_body_declarations ::=
105      aspect_body_declaration
106   |
107      aspect_body_declarations aspect_body_declaration;
108
109 aspect_body_declaration ::=
110      class_body_declaration
111   |
112      declare_declaration
```

```
113 /*|
114       pointcut_declaration*/
115     |
116       advice_declaration
117     |
118       intertype_member_declaration;
119
120 declare_declaration ::=
121       DECLARE PC_PARENTS COLON classname_pattern_expr
122              EXTENDS interface_type_list SEMICOLON
123     |
124       DECLARE PC_PARENTS COLON classname_pattern_expr
125              IMPLEMENTS interface_type_list SEMICOLON
126     |
127       DECLARE PC_WARNING COLON pointcut_expr COLON STRING_LITERAL SEMICOLON
128     |
129       DECLARE PC_ERROR COLON pointcut_expr COLON STRING_LITERAL SEMICOLON
130     |
131       DECLARE PC_SOFT COLON type COLON pointcut_expr SEMICOLON
132     |
133       DECLARE PC_PRECEDENCE COLON classname_pattern_expr_list SEMICOLON;
134
135 pointcut_declaration ::=
136       modifiers_opt POINTCUT identifier
137               LPAREN formal_parameter_list_opt RPAREN SEMICOLON
138     |
139       modifiers_opt POINTCUT identifier
140               LPAREN formal_parameter_list_opt RPAREN
141               COLON pointcut_expr SEMICOLON;
142
143 advice_declaration ::=
144        modifiers_opt advice_spec throws_opt COLON pointcut_expr
145       /* only valid modifier is strictfp */
146          method_body;
147
148 advice_spec ::=
149       BEFORE LPAREN formal_parameter_list_opt RPAREN
150     |
151       AFTER LPAREN formal_parameter_list_opt RPAREN
152     |
153       AFTER LPAREN formal_parameter_list_opt RPAREN PC_RETURNING
154     |
155       AFTER LPAREN formal_parameter_list_opt
156             RPAREN PC_RETURNING LPAREN RPAREN
```

```
157    |
158       AFTER LPAREN formal_parameter_list_opt RPAREN PC_RETURNING
159             LPAREN formal_parameter RPAREN
160    |
161       AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING
162    |
163       AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING LPAREN RPAREN
164    |
165       AFTER LPAREN formal_parameter_list_opt RPAREN PC_THROWING
166             LPAREN formal_parameter RPAREN
167    |
168       type AROUND LPAREN formal_parameter_list_opt RPAREN
169    |
170       VOID AROUND LPAREN formal_parameter_list_opt RPAREN:y
171         {: Grm.parserTrace("VOID around (formals)");
172            TypeNode voidn =  parser.nf.CanonicalTypeNode(parser.pos(a),
173                                                          parser.ts.Void());
174            Around around = parser.nf.Around(parser.pos(a,y),
175                                             voidn,
176                                             b);
177            RESULT = around;
178         :}
179    ;
180
181  intertype_member_declaration ::=
182       modifiers_opt VOID name DOT identifier
183           LPAREN formal_parameter_list_opt RPAREN throws_opt method_body
184    |
185       modifiers_opt type name DOT identifier
186           LPAREN formal_parameter_list_opt RPAREN throws_opt method_body
187    |
188       modifiers_opt name DOT NEW LPAREN formal_parameter_list_opt
189           RPAREN throws_opt constructor_body
190    |
191       modifiers_opt type name DOT identifier SEMICOLON
192    |
193       modifiers_opt type name DOT identifier
194          EQ variable_initializer SEMICOLON;
195
196  /* ------------------------ POINTCUT EXPRESSIONS ---------------------- */
197
198  pointcut_expr ::=
199      and_pointcut_expr
200   |
```

```
201     pointcut_expr PC_OROR and_pointcut_expr;
202
203 and_pointcut_expr ::=
204     unary_pointcut_expr
205   |
206     and_pointcut_expr PC_ANDAND unary_pointcut_expr;
207
208 unary_pointcut_expr ::=
209       basic_pointcut_expr
210     |
211       PC_NOT unary_pointcut_expr;
212
213 basic_pointcut_expr ::=
214      LPAREN pointcut_expr RPAREN
215     |
216      PC_CALL LPAREN method_constructor_pattern RPAREN
217     |
218      PC_EXECUTION LPAREN method_constructor_pattern RPAREN
219     |
220      PC_INITIALIZATION LPAREN constructor_pattern RPAREN
221     |
222      PC_PREINITIALIZATION LPAREN constructor_pattern RPAREN
223     |
224      PC_STATICINITIALIZATION LPAREN classname_pattern_expr RPAREN
225     |
226      PC_GET LPAREN field_pattern RPAREN
227     |
228      PC_SET LPAREN field_pattern RPAREN
229     |
230      PC_HANDLER LPAREN classname_pattern_expr RPAREN
231     |
232      PC_ADVICEEXECUTION LPAREN RPAREN
233     |
234      PC_WITHIN LPAREN classname_pattern_expr RPAREN
235     |
236      PC_WITHINCODE LPAREN method_constructor_pattern RPAREN
237     |
238      PC_CFLOW LPAREN pointcut_expr RPAREN
239     |
240      PC_CFLOWBELOW LPAREN pointcut_expr RPAREN
241     |
242      PC_IF LPAREN expression RPAREN
243     |
244      PC_THIS LPAREN type_id_star RPAREN
```

```
245     |
246       PC_TARGET LPAREN type_id_star RPAREN
247     |
248       PC_ARGS LPAREN type_id_star_list_opt RPAREN
249     |
250       name LPAREN type_id_star_list_opt RPAREN;
251
252 /* ------------------- NAME PATTERNS ---------------------------------- */
253
254 name_pattern ::=
255       simple_name_pattern
256     |
257       name_pattern DOT simple_name_pattern
258     |
259       name_pattern PC_DOTDOT simple_name_pattern;
260
261 simple_name_pattern ::=
262       PC_MULT
263     |
264       IDENTIFIERPATTERN
265     |
266       identifier
267     |
268       aspectj_reserved_identifier;
269
270 aspectj_reserved_identifier ::=
271       ASPECT
272    |
273       PRIVILEGED
274    |
275       PC_ADVICEEXECUTION
276    |
277       PC_ARGS
278    |
279       PC_CALL
280    |
281       PC_CFLOW
282    |
283       PC_CFLOWBELOW
284    |
285       PC_ERROR
286    |
287       PC_EXECUTION
288    |
```

```
289       PC_GET
290     |
291       PC_HANDLER
292     |
293       PC_INITIALIZATION
294     |
295       PC_PARENTS
296     |
297       PC_PRECEDENCE
298     |
299       PC_PREINITIALIZATION
300     |
301       PC_RETURNING
302     |
303       PC_SET
304     |
305       PC_SOFT
306     |  PC_STATICINITIALIZATION
307     |
308       PC_TARGET
309     |
310       PC_THROWING
311     |
312       PC_WARNING
313     |
314       PC_WITHINCODE;
315
316   classtype_dot_id ::=
317       simple_name_pattern
318     |
319       name_pattern DOT simple_name_pattern
320     |
321       name_pattern PC_PLUS DOT simple_name_pattern
322     |
323       name_pattern PC_DOTDOT simple_name_pattern
324     |
325       LPAREN type_pattern_expr RPAREN DOT simple_name_pattern;
326
327   classtype_dot_new ::=
328       NEW
329     |
330       name_pattern DOT NEW
331     |
332       name_pattern PC_PLUS DOT NEW
```

```
333   |
334      name_pattern PC_DOTDOT NEW
335   |
336      LPAREN type_pattern_expr RPAREN DOT NEW;
337
338 /* ------------------- TYPE PATTERNS ---------------------------------- */
339
340
341 type_pattern_expr ::=
342       or_type_pattern_expr
343    |
344       type_pattern_expr PC_ANDAND or_type_pattern_expr;
345
346 or_type_pattern_expr ::=
347       unary_type_pattern_expr
348    |
349       or_type_pattern_expr PC_OROR unary_type_pattern_expr;
350
351 unary_type_pattern_expr ::=
352       basic_type_pattern
353    |
354       PC_NOT unary_type_pattern_expr;
355
356 /* check that VOID is not in patterns for formals, ok for
357        patterns for return types */
358 basic_type_pattern ::=
359      VOID
360   |
361      base_type_pattern
362   |
363      base_type_pattern dims
364   |
365      LPAREN type_pattern_expr RPAREN;
366
367 base_type_pattern ::=
368      primitive_type
369   |
370      name_pattern
371   |
372      name_pattern PC_PLUS;
373
374 /* ------------------ CLASSNAME PATTERNS ------------------------------- */
375
376 classname_pattern_expr_list ::=
```

```
377       classname_pattern_expr
378     |
379       classname_pattern_expr_list COMMA classname_pattern_expr;
380
381   classname_pattern_expr ::=
382         and_classname_pattern_expr
383     |
384         classname_pattern_expr PC_OROR and_classname_pattern_expr;
385
386   and_classname_pattern_expr ::=
387         unary_classname_pattern_expr
388     |
389         and_classname_pattern_expr PC_ANDAND unary_classname_pattern_expr;
390
391   unary_classname_pattern_expr ::=
392         basic_classname_pattern
393     |
394         PC_NOT unary_classname_pattern_expr;
395
396   basic_classname_pattern ::=
397       name_pattern
398     |
399       name_pattern PC_PLUS
400     |
401       LPAREN classname_pattern_expr RPAREN;
402
403   classname_pattern_expr_nobang ::=
404         and_classname_pattern_expr_nobang
405     |
406         classname_pattern_expr_nobang PC_OROR and_classname_pattern_expr;
407
408   and_classname_pattern_expr_nobang ::=
409         basic_classname_pattern
410     |
411         and_classname_pattern_expr_nobang PC_ANDAND unary_classname_pattern_expr;
412
413   /* ----------------------- MODIFIER PATTERNS --------------------------- */
414
415   modifier_pattern_expr ::=
416       modifier
417     |
418       PC_NOT modifier
419     |
420       modifier_pattern_expr modifier
```

```
421   |
422       modifier_pattern_expr PC_NOT modifier;
423
424 /* -------------- METHOD, CONSTRUCTOR and FIELD PATTERNS ----------------- */
425
426 throws_pattern_list_opt ::=
427       THROWS throws_pattern_list
428   |
429       // epsilon
430        ;
431 throws_pattern_list ::=
432       throws_pattern
433   |
434       throws_pattern_list COMMA throws_pattern;
435
436 throws_pattern ::=
437       classname_pattern_expr_nobang
438   |
439       PC_NOT classname_pattern_expr;
440
441 method_constructor_pattern ::=
442       method_pattern
443   |
444       constructor_pattern;
445
446 method_pattern ::=
447       modifier_pattern_expr
448         type_pattern_expr
449         classtype_dot_id
450         LPAREN formal_pattern_list_opt RPAREN
451         throws_pattern_list_opt
452   |
453       type_pattern_expr classtype_dot_id
454         LPAREN formal_pattern_list_opt RPAREN
455         throws_pattern_list_opt;
456
457 constructor_pattern ::=
458       modifier_pattern_expr
459         classtype_dot_new
460         LPAREN formal_pattern_list_opt RPAREN
461         throws_pattern_list_opt
462   |
463         classtype_dot_new
464         LPAREN formal_pattern_list_opt RPAREN
```

```
465         throws_pattern_list_opt;
466
467 field_pattern ::=
468       modifier_pattern_expr type_pattern_expr classtype_dot_id
469     |
470       type_pattern_expr classtype_dot_id;
471
472 /* -------------------- FORMAL PARAMETER LIST PATTERNS --------------*/
473
474 formal_pattern_list_opt ::=
475       formal_pattern_list
476     |
477       // epsilon
478         ;
479 formal_pattern_list ::=
480       formal_pattern
481     |
482       formal_pattern_list COMMA formal_pattern;
483
484 formal_pattern ::=
485         PC_DOTDOT
486       |
487         DOT DOT
488       |
489         type_pattern_expr;
490
491 /* -------------------- POINTCUT PARAMETER LIST PATTERNS --------------*/
492
493 type_id_star_list_opt ::=
494       type_id_star_list
495       |
496       // epsilon
497         ;
498
499 type_id_star_list ::=
500       type_id_star
501       |
502       type_id_star_list COMMA type_id_star;
503
504 // there should be three alternatives here: star, type, and identifier
505 // disambiguation between type and identifier happens in the type-checker
506 type_id_star ::=
507         PC_MULT
508       |
```

```
509         PC_DOTDOT
510     |
511         type
512     |
513         type PC_PLUS;
```

Apêndice C

Gramática da Linguagem Java

Abaixo é apresentada a gramática da linguagem Java usada como referência neste trabalho. A gramática apresentada é a fornecida pelo desenvolvedores do Polyglot e usada para gerar o analisador sintático.

Gramática da Linguagem Java

```
1  // This parser is based on:
2  /* Java 1.2 parser for CUP.
3   * Copyright (C) 1998 C. Scott Ananian <cananian@alumni.princeton.edu>
4   * This program is released under the terms of the GPL; see the file
5   * COPYING for more details.  There is NO WARRANTY on this code.
6   *
7   * As a special exception, C. Scott Ananian additionally permits the
8   * distribution of this modified version of the parser and its derivatives
9   * under the terms of the LGPL.
10  *
11  * extended to support Java 1.4 (assert keyword).
12  */
13
14 start with goal;
15
16 // 19.2) The Syntactic Grammar
17 goal ::=
18                     // SourceFile
19         compilation_unit;
20
21 // 19.3) Lexical Structure.
22 literal ::=
23                     // Lit
24         INTEGER_LITERAL
25     |
```

```
26          LONG_LITERAL
27      |
28          DOUBLE_LITERAL
29      |
30          FLOAT_LITERAL
31      |
32          BOOLEAN_LITERAL
33      |
34          CHARACTER_LITERAL
35      |
36          STRING_LITERAL
37      |
38          NULL_LITERAL;
39  boundary_literal ::=
40                      // Lit
41          INTEGER_LITERAL_BD
42      |
43          LONG_LITERAL_BD;
44
45
46  // 19.4) Types, Values, and Variables
47  type ::=
48                      // TypeNode
49          primitive_type
50      |
51         reference_type;
52  primitive_type ::=
53                      // TypeNode
54          numeric_type
55      |
56          BOOLEAN;
57  numeric_type::=
58                      // TypeNode
59          integral_type
60      |
61         floating_point_type;
62  integral_type ::=
63                      // TypeNode
64          BYTE
65      |
66          CHAR
67      |
68          SHORT
69      |
```

```
70          INT
71      |
72          LONG;
73  floating_point_type ::=
74                      // TypeNode
75          FLOAT
76      |
77          DOUBLE;
78  reference_type ::=
79                      // TypeNode
80          class_or_interface_type
81      |
82          array_type;
83  class_or_interface_type ::=
84                      // TypeNode
85          name;
86  class_type ::=
87                      // TypeNode
88          class_or_interface_type;
89  interface_type ::=
90                      // TypeNode
91          class_or_interface_type;
92  array_type ::=
93                      // TypeNode
94          primitive_type dims
95      |
96          name dims;
97  // 19.5) Names
98
99  // Add by Leonardo
100
101 identifier ::= IDENTIFIER;
102
103 name    ::=
104                     // Name
105         simple_name
106     |
107         qualified_name;
108 simple_name ::=
109                     // Name
110         identifier;
111 qualified_name ::=
112                     // Name
113         name DOT identifier;
```

```
114 // 19.6) Packages
115 compilation_unit ::=
116                     // SourceFile
117         package_declaration_opt
118         import_declarations_opt
119         type_declarations_opt
120     |
121         error
122         type_declarations_opt;
123 package_declaration_opt ::=
124                     // PackageNode
125         package_declaration
126     |
127          ;
128 import_declarations_opt ::=
129                     // List of Import
130         import_declarations
131     |
132          ;
133 type_declarations_opt   ::=
134                     // List of TopLevelDecl
135         type_declarations
136     |
137          ;
138 import_declarations ::=
139                     // List of Import
140         import_declaration
141     |
142         import_declarations import_declaration;
143 type_declarations ::=
144                     // List of TopLevelDecl
145         type_declaration
146     |
147         type_declarations type_declaration;
148 package_declaration ::=
149                     // PackageNode
150         PACKAGE name SEMICOLON;
151 import_declaration ::=
152                     // Import
153         single_type_import_declaration
154     |
155         type_import_on_demand_declaration;
156 single_type_import_declaration ::=
157                     // Import
```

```
158         IMPORT qualified_name SEMICOLON;
159 type_import_on_demand_declaration ::=
160                     // Import
161         IMPORT name DOT MULT SEMICOLON;
162 type_declaration ::=
163                     // ClassDecl
164         class_declaration
165     |
166         interface_declaration
167     |
168         SEMICOLON;
169
170 // 19.7) Productions used only in the LALR(1) grammar
171 modifiers_opt::=
172                     // Flags
173         modifiers
174     |
175          ;
176 modifiers ::=
177                     // Flags
178         modifier
179     |
180         modifiers modifier;
181 modifier ::=
182                     // Flags
183         PUBLIC
184     |
185         PROTECTED
186     |
187         PRIVATE
188     |
189         STATIC
190     |
191         ABSTRACT
192     |
193         FINAL
194     |
195         NATIVE
196     |
197         SYNCHRONIZED
198     |
199         TRANSIENT
200     |
201         VOLATILE
```

```
202     |
203         STRICTFP;
204 // 19.8) Classes
205
206 // 19.8.1) Class Declarations
207 class_declaration ::=
208                     // ClassDecl
209         modifiers_opt CLASS identifier;
210 super ::=
211                     // TypeNode
212         EXTENDS class_type;
213 super_opt ::=
214                    // TypeNode
215         super
216     |
217          ;
218 interfaces ::=
219                     // List of TypeNode
220         IMPLEMENTS interface_type_list;
221 interfaces_opt::=
222                     // List of TypeNode
223         interfaces
224     |
225          ;
226 interface_type_list ::=
227                     // List of TypeNode
228         interface_type
229     |
230         interface_type_list COMMA interface_type;
231 class_body ::=
232                     // ClassBody
233         LBRACE class_body_declarations_opt RBRACE;
234 class_body_declarations_opt ::=
235                     // List of ClassMember
236         class_body_declarations
237     |
238          ;
239 class_body_declarations ::=
240                     // List of ClassMember
241         class_body_declaration
242     |
243         class_body_declarations class_body_declaration;
244 class_body_declaration ::=
245                     // List of ClassMember
```

```
246         class_member_declaration
247     |
248         static_initializer
249     |
250         constructor_declaration
251     |
252         block
253     |
254         SEMICOLON
255     |
256         error SEMICOLON
257     |
258         error LBRACE;
259 class_member_declaration ::=
260                     // List of ClassMember
261         field_declaration
262     |
263         method_declaration
264         /* repeat the prod for 'class_declaration' here: */
265     |
266         modifiers_opt CLASS identifier
267     |
268         interface_declaration;
269
270 // 19.8.2) Field Declarations
271 field_declaration ::=
272                     // List of ClassMember
273         modifiers_opt type variable_declarators SEMICOLON;
274 variable_declarators ::=
275                     // List of VarDeclarator
276         variable_declarator
277     |
278         variable_declarators COMMA variable_declarator;
279 variable_declarator ::=
280                 // VarDeclarator
281         variable_declarator_id
282     |
283         variable_declarator_id EQ variable_initializer;
284 variable_declarator_id ::=
285                 // VarDeclarator
286         identifier
287     |
288         variable_declarator_id LBRACK RBRACK;
289 variable_initializer ::=
```

```
290                     // Expr
291         expression
292     |
293         array_initializer;
294
295 // 19.8.3) Method Declarations
296 method_declaration ::=
297                     // MethodDecl
298         method_header method_body;
299 method_header ::=
300                     // MethodDecl
301         modifiers_opt type identifier LPAREN
302                 formal_parameter_list_opt RPAREN dims_opt throws_opt
303     |
304         modifiers_opt VOID identifier LPAREN;
305 formal_parameter_list_opt ::=
306                     // List of Formal
307         formal_parameter_list
308     |
309          ;
310 formal_parameter_list ::=
311                     // List of Formal
312         formal_parameter
313     |
314         formal_parameter_list COMMA formal_parameter;
315 formal_parameter ::=
316                     // Formal
317         type variable_declarator_id
318     |
319         FINAL type variable_declarator_id;
320 throws_opt ::=
321                     // List of TypeNode
322         throws
323     |
324          ;
325 throws ::=
326                     // List of TypeNode
327         THROWS class_type_list;
328 class_type_list ::=
329                     // List of TypeNode
330         class_type
331     |
332         class_type_list COMMA class_type;
333 method_body ::=
```

```
                    // Block
        block
    |
        SEMICOLON;

// 19.8.4) Static Initializers
static_initializer ::=
                    // Block
        STATIC block;

// 19.8.5) Constructor Declarations
constructor_declaration ::=
                    // ConstructorDecl
        modifiers_opt simple_name LPAREN formal_parameter_list_opt RPAREN
            throws_opt constructor_body;
constructor_body ::=
                    // Block
        LBRACE explicit_constructor_invocation block_statements RBRACE
    |
        LBRACE explicit_constructor_invocation RBRACE
    |
        LBRACE block_statements RBRACE
    |
        LBRACE RBRACE;
explicit_constructor_invocation ::=
                    // ConstructorCall
        THIS LPAREN argument_list_opt RPAREN SEMICOLON
    |
        SUPER LPAREN argument_list_opt RPAREN SEMICOLON
    |
        primary DOT THIS LPAREN argument_list_opt RPAREN SEMICOLON
    |
        primary DOT SUPER LPAREN argument_list_opt RPAREN SEMICOLON;

// 19.9) Interfaces

// 19.9.1) Interface Declarations
interface_declaration ::=
                    // ClassDecl
        modifiers_opt INTERFACE identifier;
extends_interfaces_opt ::=
                    // List of TypeNode
        extends_interfaces
    |
```

```
          ;
extends_interfaces ::=
                    // List of TypeNode
        EXTENDS interface_type
    |
        extends_interfaces COMMA interface_type;
interface_body ::=
                    // ClassBody
        LBRACE interface_member_declarations_opt RBRACE;
interface_member_declarations_opt ::=
                    // List of ClassMember
        interface_member_declarations
    |
          ;
interface_member_declarations ::=
                    // List of ClassMember
        interface_member_declaration
    |
        interface_member_declarations interface_member_declaration;
interface_member_declaration ::=
                    // List of ClassMember
        constant_declaration
    |
        abstract_method_declaration
    |
        class_declaration
    |
        interface_declaration
    |
        SEMICOLON;
constant_declaration ::=
                    // List of ClassMember
        field_declaration;
abstract_method_declaration ::=
                    // MethodDecl
        method_header SEMICOLON;

// 19.10) Arrays
array_initializer ::=
                    // ArrayInit
        LBRACE:n variable_initializers COMMA RBRACE
    |
        LBRACE variable_initializers RBRACE
    |
```

```
422         LBRACE COMMA RBRACE
423     |
424         LBRACE RBRACE;
425 variable_initializers ::=
426                     // List of Expr
427         variable_initializer
428     |
429         variable_initializers COMMA variable_initializer;
430
431 // 19.11) Blocks and Statements
432 block ::=
433                     // Block
434         LBRACE block_statements_opt RBRACE
435     |
436         error RBRACE;
437 block_statements_opt ::=
438                     // List of Stmt
439         block_statements
440     |
441          ;
442 block_statements ::=
443                     // List of Stmt
444         block_statement
445     |
446         block_statements block_statement;
447 block_statement ::=
448                     // List of Stmt
449         local_variable_declaration_statement
450     |
451         statement
452     |
453         class_declaration;
454 local_variable_declaration_statement ::=
455                     // List of LocalDecl
456         local_variable_declaration SEMICOLON;
457 local_variable_declaration ::=
458                     // List of LocalDecl
459         type variable_declarators
460     |   FINAL type variable_declarators;
461 statement ::=
462                     // Stmt
463         statement_without_trailing_substatement
464     |
465         labeled_statement
```

```
466     |
467         if_then_statement
468     |
469         if_then_else_statement
470     |
471         while_statement
472     |
473         for_statement
474     |
475         error SEMICOLON;
476 statement_no_short_if ::=
477                     // Stmt
478         statement_without_trailing_substatement
479     |
480         labeled_statement_no_short_if
481     |
482         if_then_else_statement_no_short_if
483     |
484         while_statement_no_short_if
485     |
486         for_statement_no_short_if;
487 statement_without_trailing_substatement ::=
488                     // Stmt
489         block
490     |
491         empty_statement
492     |
493         expression_statement
494     |
495         switch_statement:
496     |
497         do_statement
498     |
499         break_statement
500     |
501         continue_statement
502     |
503         return_statement
504     |
505         synchronized_statement
506     |
507         throw_statement
508     |
509         try_statement
```

```
510     |
511         assert_statement;
512 empty_statement ::=
513                     // Empty
514         SEMICOLON;
515 labeled_statement ::=
516                     // Labeled
517         identifier COLON statement;
518 labeled_statement_no_short_if ::=
519                     // Labeled
520         identifier COLON statement_no_short_if;
521 expression_statement ::=
522                     // Stmt
523         statement_expression SEMICOLON;
524 statement_expression ::=
525                     // Expr
526         assignment
527     |
528         preincrement_expression
529     |
530         predecrement_expression
531     |
532         postincrement_expression
533     |
534         postdecrement_expression
535     |
536         method_invocation
537     |
538         class_instance_creation_expression;
539 if_then_statement ::=
540                     // If
541         IF LPAREN expression RPAREN statement;
542 if_then_else_statement ::=
543                     // If
544         IF LPAREN expression RPAREN statement_no_short_if;
545 if_then_else_statement_no_short_if ::=
546                     // If
547         IF LPAREN expression RPAREN statement_no_short_if
548             ELSE statement_no_short_if;
549 switch_statement ::=
550                     // Switch
551         SWITCH LPAREN expression RPAREN switch_block;
552 switch_block ::=
553                     // List of SwitchElement
```

```
554         LBRACE switch_block_statement_groups switch_labels RBRACE; :}
555     |
556         LBRACE switch_block_statement_groups RBRACE
557     |
558         LBRACE switch_labels RBRACE
559     |
560         LBRACE RBRACE;
561 switch_block_statement_groups ::=
562                     // List of SwitchElement
563         switch_block_statement_group
564     |
565         switch_block_statement_groups switch_block_statement_group;
566 switch_block_statement_group ::=
567                     // List of SwitchElement
568         switch_labels block_statements;
569 switch_labels ::=
570                     // List of Case
571         switch_label
572     |
573         switch_labels switch_label;
574 switch_label ::=
575                     // Case
576         CASE constant_expression COLON
577     |
578         DEFAULT COLON;
579
580 while_statement ::=
581                     // While
582         WHILE LPAREN expression RPAREN statement;
583 while_statement_no_short_if ::=
584                     // While
585         WHILE LPAREN expression RPAREN statement_no_short_if;
586 do_statement ::=
587                     // Do
588         DO statement WHILE LPAREN expression RPAREN SEMICOLON;
589 for_statement ::=
590                     // For
591         FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
592             for_update_opt RPAREN statement;
593 for_statement_no_short_if ::=
594                     // For
595         FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
596             for_update_opt RPAREN statement_no_short_if;
597 for_init_opt ::=
```

```
598                     // List of ForInit
599         for_init
600     |
601          ;
602 for_init ::=
603                     // List of ForInit
604         statement_expression_list
605     |
606         local_variable_declaration;
607 for_update_opt ::=
608                     // List of ForUpdate
609         for_update
610     |
611          ;
612 for_update ::=
613                     // List of ForUpdate
614         statement_expression_list;
615 statement_expression_list ::=
616                     // List of Stmt
617         statement_expression
618     |
619         statement_expression_list COMMA statement_expression;
620
621 identifier_opt ::=
622                     // Name
623         identifier
624     |
625          ;
626
627 break_statement ::=
628                     // Branch
629         BREAK identifier_opt SEMICOLON;
630
631 continue_statement ::=
632                     // Branch
633         CONTINUE identifier_opt SEMICOLON;
634 return_statement ::=
635                     // Return
636         RETURN expression_opt SEMICOLON;
637 throw_statement ::=
638                     // Throw
639         THROW expression SEMICOLON;
640 synchronized_statement ::=
641                     // Synchronized
```

```
642         SYNCHRONIZED LPAREN expression RPAREN block;
643 try_statement ::=
644                     // Try
645         TRY block catches
646     |
647         TRY block catches_opt finally;
648 catches_opt ::=
649                     // List of Catch
650         catches
651     |
652          ;
653 catches ::=
654                     // List of Catch
655         catch_clause
656     |
657         catches catch_clause;
658 catch_clause ::=
659                     // Catch
660         CATCH LPAREN formal_parameter RPAREN block;
661 finally ::=
662                     // Block
663         FINALLY block;
664
665 assert_statement ::=
666                     // Assert
667         ASSERT expression SEMICOLON
668     |
669         ASSERT expression COLON expression SEMICOLON;
670
671 // 19.12) Expressions
672 primary ::=
673                     // Expr
674         primary_no_new_array
675     |
676         array_creation_expression;
677 primary_no_new_array ::=
678                     // Expr
679         literal
680     |
681         THIS
682     |
683         LPAREN expression RPAREN
684     |
685         class_instance_creation_expression
```

```
686      |
687          field_access
688      |
689          method_invocation
690      |
691          array_access
692      |
693          primitive_type DOT CLASS
694      |
695          VOID DOT CLASS
696      |
697          array_type DOT CLASS
698      |
699          name DOT CLASS
700      |
701          name DOT THIS;
702  class_instance_creation_expression ::=
703                       // Expr
704          NEW class_type LPAREN argument_list_opt RPAREN
705      |
706          NEW class_type LPAREN argument_list_opt RPAREN class_body
707      |
708          primary DOT NEW simple_name LPAREN argument_list_opt RPAREN
709      |
710          primary DOT NEW simple_name LPAREN argument_list_opt RPAREN class_body
711      |
712          name DOT NEW simple_name LPAREN argument_list_opt RPAREN
713      |
714          name DOT NEW simple_name LPAREN argument_list_opt RPAREN class_body;
715  argument_list_opt ::=
716                       // List of Expr
717          argument_list
718      |
719           ;
720  argument_list ::=
721                       // List of Expr
722          expression
723      |
724          argument_list COMMA expression;
725  array_creation_expression ::=
726                       // NewArray
727          NEW primitive_type dim_exprs dims_opt
728      |
729          NEW class_or_interface_type dim_exprs dims_opt
```

```
730     |
731         NEW primitive_type dims array_initializer
732     |
733         NEW class_or_interface_type dims array_initializer;
734 dim_exprs ::=
735                     // List of Expr
736         dim_expr
737     |
738         dim_exprs dim_expr;
739 dim_expr ::=
740                     // Expr
741         LBRACK:x expression RBRACK;
742 dims_opt ::=
743                     // Integer
744         dims
745     |
746          ;
747 dims ::=
748                     // Integer
749         LBRACK RBRACK
750     |
751         dims LBRACK RBRACK;
752 field_access ::=
753                     // Field
754         primary DOT identifier
755     |
756         SUPER DOT identifier
757     |
758         name DOT SUPER DOT identifier;
759 method_invocation ::=
760                     // Call
761         name LPAREN argument_list_opt RPAREN
762     |
763         primary DOT identifier LPAREN argument_list_opt RPAREN
764     |
765         SUPER DOT identifier LPAREN argument_list_opt RPAREN
766     |
767         name DOT SUPER DOT identifier LPAREN argument_list_opt RPAREN;
768 array_access ::=
769                     // ArrayAccess
770         name LBRACK expression RBRACK
771     |
772         primary_no_new_array LBRACK expression RBRACK;
773 postfix_expression ::=
```

```
774                     // Expr
775         primary
776     |
777         name
778     |
779         postincrement_expression
780     |
781         postdecrement_expression;
782 postincrement_expression ::=
783                     // Unary
784         postfix_expression PLUSPLUS;
785 postdecrement_expression ::=
786                     // Unary
787         postfix_expression MINUSMINUS;
788 unary_expression ::=
789                     // Expr
790         preincrement_expression
791     |
792         predecrement_expression
793     |
794         PLUS unary_expression
795     |
796         MINUS unary_expression
797     |
798         MINUS boundary_literal
799     |
800         unary_expression_not_plus_minus;
801 preincrement_expression ::=
802                     // Unary
803         PLUSPLUS unary_expression;
804 predecrement_expression ::=
805                     // Unary
806         MINUSMINUS unary_expression;
807 unary_expression_not_plus_minus ::=
808                     // Expr
809         postfix_expression
810     |
811         COMP unary_expression
812     |
813         NOT unary_expression
814     |
815         cast_expression;
816 cast_expression ::=
817                     // Cast
```

```
818         LPAREN primitive_type dims_opt RPAREN unary_expression
819     |
820         LPAREN expression RPAREN unary_expression_not_plus_minus
821     |
822         LPAREN name dims RPAREN unary_expression_not_plus_minus;
823 multiplicative_expression ::=
824                     // Expr
825         unary_expression
826     |
827         multiplicative_expression MULT unary_expression
828     |
829         multiplicative_expression DIV unary_expression
830     |
831         multiplicative_expression MOD unary_expression;
832 additive_expression ::=
833                     // Expr
834         multiplicative_expression
835     |
836         additive_expression PLUS multiplicative_expression
837     |
838         additive_expression MINUS multiplicative_expression;
839 shift_expression ::=
840                     // Expr
841         additive_expression
842     |
843         shift_expression LSHIFT additive_expression
844     |
845         shift_expression RSHIFT additive_expression
846     |
847         shift_expression URSHIFT additive_expression;
848 relational_expression ::=
849                     // Expr
850         shift_expression
851     |
852         relational_expression LT shift_expression
853     |
854         relational_expression GT shift_expression
855     |
856         relational_expression LTEQ shift_expression
857     |
858         relational_expression GTEQ shift_expression
859     |
860         relational_expression INSTANCEOF reference_type;
861
```

```
862 equality_expression ::=
863                     // Expr
864         relational_expression
865     |
866         equality_expression EQEQ relational_expression
867     |
868         equality_expression NOTEQ relational_expression;
869 and_expression ::=
870                     // Expr
871         equality_expression
872     |
873         and_expression AND equality_expression;
874 exclusive_or_expression ::=
875                     // Expr
876         and_expression
877     |
878         exclusive_or_expression XOR and_expression;
879 inclusive_or_expression ::=
880                     // Expr
881         exclusive_or_expression
882     |
883         inclusive_or_expression OR exclusive_or_expression;
884 conditional_and_expression ::=
885                     // Expr
886         inclusive_or_expression
887     |
888         conditional_and_expression ANDAND inclusive_or_expression;
889 conditional_or_expression ::=
890                     // Expr
891         conditional_and_expression
892     |
893         conditional_or_expression OROR conditional_and_expression;
894 conditional_expression ::=
895                     // Expr
896         conditional_or_expression
897     |
898         conditional_or_expression QUESTION expression;
899 assignment_expression ::=
900                     // Expr
901         conditional_expression
902     |
903         assignment;
904 assignment ::=
905                     // Expr
```

```
906         left_hand_side assignment_operator assignment_expression;
907 left_hand_side ::=
908                     // Expr
909         name
910     |
911         field_access
912     |
913         array_access;
914 assignment_operator ::=
915                     // Assign.Operator
916         EQ
917     |
918         MULTEQ
919     |
920         DIVEQ
921     |
922         MODEQ
923     |
924         PLUSEQ
925     |
926         MINUSEQ
927     |
928         LSHIFTEQ
929     |
930         RSHIFTEQ
931     |
932         URSHIFTEQ
933     |
934         ANDEQ
935     |
936         XOREQ
937     |
938         OREQ;
939 expression_opt ::=
940                     // Expr
941         expression
942     |
943          ;
944 expression ::=
945                     // Expr
946         assignment_expression;
947 constant_expression ::=
948                     // Expr
949         expression;
```

Referências Bibliográficas

Allen, E.; Chase, D.; Hallett, J.; Luchangco, V.; Maessen, J.-W.; Ryu, S.; Jr., G. L. S. & Tobin-Hochstadt, S. (2008). The Fortress Language Specification. Technical report, Sun Microsystems, Inc.

Allen, E.; Culpepper, R. & Nielsen, J. D. (2009). Growing a syntax. In *Foundations of Object-Oriented Languages (FOOL'2009)*, pp. 1–11, New York, NY, USA. ACM.

Aske, P. A.; Avgustinov, P.; Christensen, A. S.; Hendren, L.; Kuzins, S.; Lhoták, J.; Moor, O. D.; Sereni, D.; Sittampalam, G. & Tibble, J. (2005). abc : An extensible AspectJ compiler. In *In AOSD 05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pp. 87–98. ACM Press.

AspectJ, E. The AspectJ Compiler. Avaliable from URL: http://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html - Acessado pela última vez em 2009.

AspectJ, E. The AspectJ Eclipse Project. Avaliable from URL: http://eclipse.org/aspectj - Acessado pela última vez em 2009.

Avgustinov, P.; Christensen, A. S.; Hendren, L.; Kuzins, S.; Lhoták, J.; Lhoták, O.; de Moor, O.; Sereni, D.; Sittampalam, G. & Tibble, J. (2005). abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pp. 87–98, New York, NY, USA. ACM.

Avgustinov, P.; Ekman, T. & Tibble, J. (2008). Modularity first: a case for mixing AOP and attribute grammars. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pp. 25–35, New York, NY, USA. ACM.

Bachrach, J. & Playford, K. (2001). The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Pro-*

gramming, Systems, Languages, and Applications, pp. 31–42, New York, NY, USA. ACM.

Bagge, A. H. & Kalleberg, K. T. (2006). DSAL = library+notation: Program Transformation for Domain-Specific Aspect Languages. In *Proceedings of the Domain-Specific Aspect Languages Workshop*, pp. 1–8.

Baker, J. & Hsieh, W. C. (2002). Maya: multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pp. 270–281, New York, NY, USA. ACM.

Batory, D.; Lofaso, B. & Smaragdakis, Y. (1998). JTS: Tools for implementing domain-specific languages. In *5th International Conference on Software Reuse*, pp. 143–153. IEEE.

Bravenboer, M.; de Groot, R. & Visser, E. (2006a). MetaBorg in action: Examples of domain-specific language embedding and assimilation using Stratego/XT. In Lämmel, R. & Saraiva, J., editores, *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *Lecture Notes in Computer Science*, pp. 297–311, Braga, Portugal. Springer Verlag.

Bravenboer, M.; Kalleberg, K. T.; Vermaas, R. & Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70.

Bravenboer, M.; Tanter, E. & Visser, E. (2006b). Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-lr parsing. In Cook, W. R., editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06)*, pp. 209–228, Portland, Oregon, USA. ACM Press.

Bravenboer, M. & Visser, E. (2004). Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In Schmidt, D. C., editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)*, pp. 365–383, Vancouver, Canada. ACM Press.

Breuel, C. & Reverbel, F. (2007). Join point selectors. In *SPLAT '07: Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies*, p. 3, New York, NY, USA. ACM.

Brukman, M. & Myers, A. C. (2008). PPG: A parser generator for extensible grammars. http://www.cs.cornell.edu/Projects/polyglot/ppg.html. Último acesso em 2009.

Cheatham, Jr., T. E. (1969). Motivation for extensible languages. *SIGPLAN Not.*, 4(8):45–49.

Chiba, S. & Nakagawa, K. (2004). Josh: an open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pp. 102–111, New York, NY, USA. ACM.

Christensen, C. & Shaw, C. J. (1969). Proceedings of the extensible languages symposium.

Clark., T. (2009). First class grammars for language oriented programming. In *The 13th World Multi-Conference on Systemics, Cybernetics and Informatics: WMSCI 2009*.

Clark, T.; Sammut, P. & Willans, J. (2008). Beyond Annotations: A Proposal for Extensible Java (XJ). *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:229–238.

Clinger, W. & Rees, J. (1991). Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 155–162, New York, NY, USA. ACM.

de Rauglaudre, D. (2009). Camlp5. Publish at http://cristal.inria.fr/ ddr/camlp5/.

Deursen, A.; Klint, P. & Visser, J. (2000). Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36.

Di Iorio, V. O.; Goulart, C. C.; Reis, L. V. S. & Oikawa, M. (2008). An Application-Specific Language for Synchronization Using Aspect-Oriented Programming Concepts. In *Proceedings of the II Latin American Workshop on Aspect-Oriented Software Development*, pp. 70–79. Institute of Computing, UNICAMP.

Di Iorio, V. O.; Reis, L. V. d. S.; Bigonha, R. d. S. & Bigonha, M. A. d. S. (2009). A proposal for extensible AspectJ. In *DSAL '09: Proceedings of the 4th workshop on Domain-specific aspect languages*, pp. 21–24, New York, NY, USA. ACM.

Dmitriev, S. (2004). Language Oriented Programming: The Next Programming Paradigm. *onBoard Online Magazine*, 1.

Duby, J. J. (1971). Extensible languages: A potential user's point of view. *SIGPLAN Not.*, 6(12):137–140.

Dybvig, R. K. (2009). *The Scheme Programming Language, 4th Edition.* The MIT Press.

Ekman, T. & Hedin, G. (2007). The JastAdd extensible Java compiler. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, pp. 1–18.

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122.

Fowler, M. (2005). Language Workbenches: The Killer-App for Domain Specific Languages? Last update in June 2005 at http://martinfowler.com/articles/languageWorkbench.html.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, Bookman.

Grimm, R. (2006). Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 38–51, New York, NY, USA. ACM.

Hannemann, J. & Kiczales, G. (2002). Design pattern implementation in Java and aspectJ. *SIGPLAN Not.*, 37(11):161–173.

Haungs, J. (1998). Oopsla '98 addendum: Addendum to the 1998 proceedings of the conference on object-oriented programming, systems, languages, and applications (addendum).

Heering, J.; Hendriks, P. R. H.; Klint, P. & Rekers, J. (1989). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75.

Huang, S. S. & Smaragdakis, Y. (2006). Easy language extension with meta-aspectj. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pp. 865–868, New York, NY, USA. ACM.

Huang, S. S.; Zook, D. & Smaragdakis, Y. (2008). Domain-specific languages and program generation with meta-AspectJ. *ACM Trans. Softw. Eng. Methodol.*, 18(2):1–32.

Hudson, S. E.; Flannery, F.; Ananian, C. S.; Wang, D. & Appel, A. (1996). CUP LALR parser generator for java. Located at http://www.cs.princeton.edu/ appel/modern/java/CUP/. Último acesso em 2009.

Ierusalimschy, R.; de Figueiredo, L. H. & Filho, W. C. (1996). Lua—an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. & Griswold, W. G. (2001). An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 327–353, London, UK. Springer-Verlag.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; marc Loingtier, J. & Irwin, J. (1997). Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, pp. 220–242. Springer-Verlag.

Kieburtz, R. B.; McKinney, L.; Bell, J. M.; Hook, J.; Kotov, A.; Lewis, J.; Oliva, D. P.; Sheard, T.; Smith, I. & Walton, L. (1996). A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pp. 542–552, Washington, DC, USA. IEEE Computer Society.

Knöll, R. & Mezini, M. (2009). π: a pattern language. In *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pp. 503–522, New York, NY, USA. ACM.

Kohlbecker, E.; Friedman, D. P.; Felleisen, M. & Duba, B. (1986). Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 151–161, New York, NY, USA. ACM.

Korenjak, A. J. (1969). Efficient LR(1) processor construction. In *STOC '69: Proceedings of the First Annual ACM Symposium on Theory of Computing*, pp. 191–200, New York, NY, USA. ACM.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA.

Lorenz, D. H. (1998). Visitor Beans: An Aspect-Oriented Pattern. In Demeyer, S. & Bosch, J., editores, *ECOOP Workshops*, volume 1543 of *Lecture Notes in Computer Science*, pp. 431–432. Springer.

Mernik, M.; Heering, J. & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Nystrom, N.; Clarkson, M. R. & Myers, A. C. (2003). Polyglot: An Extensible Compiler Framework for Java. In *In 12th International Conference on Compiler Construction*, pp. 138–152. Springer-Verlag.

Reis, L. V. d. S.; Di Iorio, V. O.; Bigonha, R. d. S.; Bigonha, M. A. d. S. & Ladeira, R. d. C. (2009). XAJ: An extensible aspect-oriented language. In *Proceedings of the III Latin American Workshop on Aspect-Oriented Software Development*, pp. 57–62. Federal University of CearÃ¡.

Riehl, J. (2006). Assimilating MetaBorg:: embedding language tools in languages. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pp. 21–28, New York, NY, USA. ACM.

Sammet, J. E. (1971). Application of extensible languages to specialized application languages. *SIGPLAN Not.*, 6(12):141–143.

Schuman, S. A. (1971). Proceedings of the international symposium on extensible languages.

Schuman, S. A. & Jorrand, P. (1970). Definition mechanisms in extensible programming languages. In *AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, fall joint computer conference*, pp. 9–20, New York, NY, USA. ACM.

Sheard, T. & Jones, S. P. (2002). Template meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75.

Shonle, M.; Lieberherr, K. & Shah, A. (2003). Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 28–37, New York, NY, USA. ACM.

Standish, T. A. (1975). Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21.

Steele, Jr., G. L. (1998). Growing a language. In *OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pp. 221–236, New York, NY, USA. ACM.

Steele Jr., G. L., editor (1990). *Common Lisp: The Language*. Digital Press, second edição.

Tatsubori, M.; Chiba, S.; Itano, K. & Killijian, M.-O. (2000). OpenJava: A Class-Based Macro System for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pp. 117–133, London, UK. Springer-Verlag.

van Deursen, A. & Klint, P. (1997). Little Languages : Little Maintenance? cwireport R 9704, CWI.

Visser, E. (1997a). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam.

Visser, E. (1997b). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam.

Visser, E. (2001). Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA 01), volume 2051 of Lecture Notes in Computer Science*, pp. 357–361. Springer-Verlag.

Vogt, H. H.; Swierstra, S. D. & Kuiper, M. F. (1989). Higher order attribute grammars. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pp. 131–145, New York, NY, USA. ACM.

Ward, M. P. (1994). Language Oriented Programming. *Software/Concepts and Tools*, 15:147–161.

Wyk, E. V. (2003). Aspects as modular language extensions. *Electronic Notes in Theoretical Computer Science*, 82(3):555 – 574. LDTA'2003 - Language descriptions, Tools and Applications.

Wyk, E. V. (2007). Implementing aspect-oriented programming constructs as modular language extensions. *Sci. Comput. Program.*, 68(1):38–61.

Wyk, E. V.; Bodin, D.; Gao, J. & Krishnan, L. (2008). Silver: an Extensible Attribute Grammar System. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116.

Wyk, E. V.; de Moor, O.; Backhouse, K. & Kwiatkowski, P. (2002). Forwarding in attribute grammars for modular language design. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pp. 128–142, London, UK. Springer-Verlag.