FELIPE DA SILVA PINHEIRO

METAPROGRAMAÇÃO EXTENSÍVEL PARA A LINGUAGEM XAJ (EXTENSIBLE ASPECTJ)

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

Viçosa
Minas Gerais - Brasil
2013

Ficha catalográfica preparada pela Seção de Catalogação e Classificação da Biblioteca Central da UFV

T
P654m
2013

Pinheiro, Felipe da Silva, 1988-
Metaprogramação extensível para a linguagem XAJ (eXtensible AspectJ) / Felipe da Silva Pinheiro. – Viçosa, MG, 2013.
x, 33f. : il. (algumas color.) ; 29cm.

Orientador: Vladimir Oliveira Di Iorio
Dissertação (mestrado) - Universidade Federal de Viçosa.
Referências bibliográficas: f. 31-33

1. Linguagem de programação (Computadores).
2. Programação (Computadores). I. Universidade Federal de Viçosa. Departamento de Informática. Programa de Pós-Graduação em Ciência da Computação. II. Título.

CDD 22. ed. 005.13

FELIPE DA SILVA PINHEIRO

METAPROGRAMAÇÃO EXTENSÍVEL PARA A LINGUAGEM XAJ (EXTENSIBLE ASPECTJ)

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

APROVADA: 26 de fevereiro de 2013.

Alcione de Paiva Oliveira
(Coorientador)

Leacir Nogueira Bastos

Vladimir Oliveira Di Iorio
(Orientador)

Aos meus pais, Luiz Alberto e Selma, pelo incentivo, pela educação e pelos valores a mim passados.

"The limits of my language mean the limits of my world."
(Ludwig Wittgenstein)

AGRADECIMENTOS

A Deus, por iluminar meu caminho em mais uma jornada que chega ao seu fim.
À Universidade Federal de Viçosa e ao Departamento de Informática, pela oportunidade de cursar o programa de pós-graduação oferecido.
À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), pela concessão da bolsa de mestrado.
Ao Prof. Vladimir, por me orientar neste trabalho, com paciência e compreensão, e por me motivar e ajudar quando preciso.
Aos meus pais, Luiz Alberto e Selma, por me incentivarem com carinho e paciência.
Aos meus familiares que estão sempre presentes na minha vida.
Aos meus amigos que sempre me animam, nunca me deixando esquecer que a diversão também é importante.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

PINHEIRO, Felipe da Silva, M.Sc., Universidade Federal de Viçosa, fevereiro de 2013. **Metaprogramação extensível para a linguagem XAJ (eXtensible AspectJ).** Orientador: Vladimir Oliveira Di Iorio. Coorientadores: Alcione de Paiva Oliveira e Ricardo dos Santos Ferreira.

Linguagens de domínio específico orientadas a aspectos (*DSALs*) são linguagens projetadas especificamente para um domínio com funcionalidades orientadas a aspectos, gerando vantagens como produtividade e expressividade. Embarcar uma *DSAL* em uma linguagem de propósito geral orientada a aspectos pode apresentar algumas vantagens. Para conseguir isso, uma das técnicas utilizadas é a extensibilidade. A linguagem *eXtensible AspectJ* (*XAJ*) é uma linguagem extensível baseada em *AspectJ*, principal linguagem orientada a aspectos. A definição de extensões para a linguagem é modular, sendo encapsulada em uma *classe sintática*, que permite definir a sintaxe e a semântica da nova construção, além de representá-la na *árvore de sintaxe abstrata* (*AST*). A definição da semântica de uma nova construção se dá pela tradução para construções já conhecidas. Nesse ponto, a linguagem pode-se beneficiar das vantagens da *metaprogramação*. A metaprogramação é uma técnica de escrever programas que geram ou manipulam programas. Assim, com uma sintaxe próxima à sintaxe concreta da linguagem, obtém-se maior facilidade de entendimento. Com um mecanismo de metaprogramação extensível, pode-se utilizar construções definidas pelo usuário na definição de novas construções. Este trabalho apresenta um mecanismo de metaprogramação extensível para a linguagem *XAJ*, apresentando exemplo de uso e suas vantagens.

Abstract

PINHEIRO, Felipe da Silva, M.Sc., Universidade Federal de Viçosa, February of 2013.
Extensible metaprogramming for the XAJ (eXtensible AspectJ) language.
Adviser: Vladimir Oliveira Di Iorio. Co-advisers: Alcione de Paiva Oliveira and Ricardo dos Santos Ferreira.

Domain specific aspect languages (*DSALs*) are languages specifically designed for a domain with aspect-oriented features, obtaining benefits such as productivity and expressiveness. To embed a *DSAL* in an aspect-oriented general purpose language may present some advantages. To achieve this, one of the techniques used is language extensibility. *XAJ* (*eXtensible AspectJ*) is an extensible language based on *AspectJ*, the main aspect-oriented language. The definitions of extensions to the language are modular, being encapsulated in a *syntax class*. Syntax class allow users define the syntax and semantics of a new construction, and represent it in the *abstract syntax tree* (*AST*). The definition of the semantics of a new construction is given by the translation to already known constructs. At this point, the language can benefit from the advantages of *metaprogramming*. Metaprogramming is a technique of writing programs that generate or manipulate other programs. Metaprogramming commands are very ease to understand especially when they use a syntax that is very similar to the concrete syntax of the language. With an extensible metaprogramming mechanism, user defined constructs can be used in the definitions of new constructs. This work presents an extensible metaprogramming mechanism for the *XAJ* language, presenting examples of usage and its advantages.

Capítulo 1

Introdução

Linguagens de Domínio Específico (*DSL*, do inglês *Domain-Specific Languages*) são linguagens de programação desenvolvidas levando em conta os problemas e as notações de um domínio específico. Assim sendo, são melhores para expressar as soluções do domínio em questão (van Deursen et al., 2000). Com o uso de *DSLs*, há um ganho de produtividade, expressividade e facilidade de uso (Mernik et al., 2005).

Linguagens de Domínio Específico Orientadas a Aspectos (*DSAL*, do inglês *Domain-Specific Aspect Languages*) são *DSLs* envolvendo o paradigma de orientação a aspectos (Kiczales et al., 1997). A *programação orientada a aspectos* surgiu em 1997, tendo como principal objetivo apresentar solução adequada para a implementação de interesses transversais. Interesses transversais são funcionalidades do sistema que não podem ser adequadamente encapsuladas, tendo, por isto, o seu código espalhado pelo sistema. Esse paradigma define uma nova unidade de compilação denominada *aspecto*, que permite especificar um código que altere outras partes do código do programa. Assim, torna-se possível a implementação modular de interesses transversais.

Embarcar uma *DSL* em uma linguagem de propósito geral apresenta algumas vantagens (Batory et al., 1998). Para realizar essa tarefa uma das técnicas aplicadas, às linguagens de propósito geral, é a extensibilidade. Linguagem extensível permite estender sua própria sintaxe concreta. A linguagem *XAJ* (*eXtensible AspectJ*) (Di Iorio et al., 2009) é uma linguagem extensível orientada a aspectos, que permite obter-se as vantagens de *DSL* embarcada, além dos benefícios da programação orientada a aspectos.

XAJ é uma linguagem que permite estender a sintaxe concreta de *AspectJ* (Laddad, 2003), principal linguagem orientada a aspectos. Em *XAJ*, pode se definir a sintaxe e a semântica das novas construções a serem incorporadas à linguagem.

Como exemplo, podemos definir um comando *fae* (uma abreviação para *"find*

and execute" que, em português, seria "achar e executar") que percorre uma coleção executando comandos quando encontra elementos que satisfaçam certa condição, ou executa comandos quando não encontra nenhum elemento. O uso de tal comando fica conforme apresentado na Figura 1.1.

```
1  ...
2  List people;
3  ...
4  fae Person p from people
5  where p.age < 18 {
6     return p;
7  } else {
8     return null;
9  }
10 ...
```

Figura 1.1. Uso do comando *fae* proposto.

O comando mostrado na Figura 1.1 percorre uma coleção *people*, contendo elementos do tipo *Person*, e, ao achar um elemento com o atributo *age* menor que 18, retorna o elemento; caso não encontre nenhum elemento, simplesmente, retorna *null*. O identificador *p* serve apenas como pseudônimo do elemento que está sendo testado diante da condição.

A forma utilizada em XAJ, para se definir a semântica, é uma tradução para construções já conhecidas, podendo executar operações sobre a *árvore de sintaxe abstrata* (*AST*, do inglês *abstract syntax tree*). Uma técnica que pode facilitar a construção de uma *AST* para a tradução é a metaprogramação com uma sintaxe próxima à sintaxe concreta da linguagem. Metaprogramação, ou programação gerativa, é a técnica de escrever programas que geram programas. Em Reis (2010), sugere-se que a linguagem XAJ pode se beneficiar do uso de metaprogramação para a tradução de novas construções, o que serviu de motivação para o desenvolvimento deste trabalho.

Na Figura 1.2, é mostrado um exemplo em alto nível da definição do comando *fae*, para melhor visualização do problema. A definição contém duas partes. A primeira parte apresenta a sintaxe para o uso do novo comando. Os elementos entre aspas são interpretados como terminais da linguagem e devem aparecer exatamente como descritos.: *"fae"*, *"from"*, *"where"* e *"else"*. Os demais elementos são interpretados como símbolos não terminais de uma especificação sintática. Assim, *tipo*, *id*, *expr* e *bloco* representam não terminais que derivam uma declaração de tipo, um identificador, expressões e blocos de comandos, respectivamente. Os índices (como em *expr_1* e *expr_2* servem para diferenciar dois símbolos iguais, dentro da mesma definição.

Pode-se observar uma correspondência entre essa definição sintática e o exemplo de uso do comando *fae* apresentado na Figura 1.1.

A segunda parte da definição da Figura 1.2 estabelece como a nova construção pode ser traduzida, usando apenas elementos da linguagem base (neste caso, apenas elementos de *Java* puro). Observe que essa tradução é apresentada como se fosse um modelo (*template*) onde alguns elementos devem ser substituídos por valores instanciados, quando a definição sintática "casar" com o programa fonte. Por exemplo, *%tipo* seria substituído por *Person*, quando o comando da Figura 1.1 fosse analisado. É exatamente nessa tradução que entra em ação a metaprogramação. Pode-se observar que se trata de geração de um novo trecho de programa, e o fato de poder usar uma sintaxe concreta bem próxima da sintaxe real da linguagem base torna o processo bem mais fácil e legível.

```
1  definição fae{
2     definição sintática{
3        ‘‘fae’’ tipo id ‘‘from’’ expr_1 ‘‘where’’ expr_2 bloco_1
4        (‘‘else’’ bloco_2)?
5     }
6     definição semântica{
7        %tipo %id;
8        boolean done = false;
9        for ( Iterator it = %expr_1. iterator (); it. hasNext ();)
10       { %id = it. next ();
11         if (%expr_2) {
12            done = true ;
13            %bloco_1
14         }
15       }
16       if (!done )
17         %bloco_2
18    }
19 }
```

Figura 1.2. Exemplo de definição do comando *fae* em alto nível.

As classes sintáticas da linguagem *XAJ* utilizam uma estrutura bem parecida com o esquema apresentado na Figura 1.2, para definir uma nova construção para a linguagem. Recursos de metaprogramação são usados para especificar a tradução da nova construção. O problema abordado neste trabalho é que, para oferecer facilidades como o uso de uma sintaxe concreta bem próxima da sintaxe da linguagem base, o sistema restringe os elementos que podem ser usados na metaprogramação, permitindo apenas o uso de estruturas predefinidas na linguagem. Mas, isso dificulta o uso de uma estrutura definida pelo programador na definição semântica de uma nova construção. Por exemplo, tendo o comando *fae*, apresentado nesta seção, pode-se construir um comando *select*. Segue um exemplo de seu uso:

```
...
List people;
...
select Person p from people
 where p.age < 18;
...
```

A semântica, neste caso, é percorrer uma coleção *people* com elementos do tipo *Person*, adicionando os elementos que satisfaçam a condição $p.age < 18$ a uma lista, retornando-a, ou, caso não haja elementos que satisfaçam a condição, retorne *null*. Como no caso do comando *fae*, o identificador p serve apenas como pseudônimo para o elemento a ser testado. Essa semântica poderia ser escrita com o seguinte código:

```
List<%tipo> res;
fae %tipo %id from %expr_1
where %expr_2 {
   res.add(%id);
}
else {
   res = null;
}
return res;
```

No código acima, *%tipo* estaria associado a *Person*, *%id* estaria associado a p, *%expr_1* estaria associado a *people* e *%expr_2* estaria associado a $p.age < 18$, para o exemplo de uso do comando *select* apresentado anteriormente. Mas, sem um mecanismo extensível, não seria possível a utilização do comando *fae* na metaprogramação, para descrever a semântica da nova construção, fazendo assim uma diferenciação entre construções nativas da linguagens e as construídas pelo programador. Além disso, a descrição da semântica torna-se mais trabalhosa, sem o uso da metaprogramação.

Neste trabalho, é apresentado um mecanismo para a utilização de metaprogramação extensível na linguagem *XAJ*. Com esse mecanismo, consegue-se uma facilidade maior na construção do código e as novas construções podem ser usadas da mesma forma que as construções nativas.

Além disso, uma sintaxe próxima à sintaxe concreta da linguagem facilita o aprendizado do mecanismo. Sem a metaprogramação, para se criar uma *AST*, é preciso criar os nós pela utilização do seu construtor ou o padrão Fábrica (Gamma et al., 1995), se disponível. Assim, com um mecanismo de metaprogramação, obtém-se mais vantagens, como a independência de implementação, aumentando a portabilidade do código, e também os detalhes de implementação ficam escondidos do programador.

1.1 Hipótese

Um mecanismo de metaprogramação extensível para a linguagem *XAJ* pode ajudar no entendimento dos programas escritos nessa linguagem e no aprendizado da mesma, pois, com uma sintaxe próxima à sintaxe concreta, não é necessário conhecer e manipular as classes que compõem a árvore de sintaxe abstrata.

1.2 Objetivos

O objetivo principal desse trabalho é prover um mecanismo para suporte à metaprogramação extensível para a linguagem *XAJ* (*eXtensible AspectJ*).

Os objetivos secundários a serem alcançados são:

a) Facilitar a construção de *AST* para a tradução de novos comandos em *XAJ*, sem diferenciar entre construções nativas e definidas pelo programador.

b) Prover uma sintaxe concreta próxima à sintaxe concreta da linguagem, facilitando o aprendizado do mecanismo.

c) Tornar o código independente de implementação, escondendo seus detalhes.

1.3 Metodologia

O trabalho foi realizado seguindo os seguintes passos:

Passo 1: Escolher um exemplo para uma extensão em *XAJ*.

Passo 2: Realizar manualmente as alterações para a extensão proposta, a fim de entender melhor o compilador *MAJ*.

Passo 3: Estender a gramática de *MAJ* para reconhecer classes sintáticas.

Passo 4: Montar as tabelas de equivalências de não terminais para todas as gramáticas envolvidas no trabalho.

Passo 5: Desenvolver um mecanismo para estender as gramáticas necessárias.

Passo 6: Desenvolver um mecanismo para extrair informações das classes sintáticas para elaborar as extensões necessárias.

Passo 7: Integrar os dois mecanismos desenvolvidos nos passos anteriores.

Passo 8: Testar o mecanismo desenvolvendo uma extensão em *XAJ*, podendo ser usado o exemplo inicialmente proposto, a fim de comparar o processo manual com o automático.

1.4 Estrutura do documento

No Capítulo 2, são apresentados trabalhos relacionados ao tema desta dissertação. São abordadas a programação orientada a aspectos, com uma breve apresentação de *AspectJ*, a metaprogramação, citando algumas ferramentas que a utilizam, além das duas ferramentas ligadas diretamente à dissertação, *Meta-AspectJ* e *XAJ*. No Capítulo 3, descreve-se um exemplo de motivação, o sistema desenvolvido, seu processamento e automatização, além de uma breve avaliação do trabalho desenvolvido. No Capítulo 4, são apresentadas as conclusões, as vantagens do trabalho e as dificuldades encontradas durante o desenvolvimento, bem como propostas de trabalhos futuros.

Capítulo 2

Revisão Bibliográfica

Como explicado no Capítulo 1, o objetivo principal desta pesquisa é prover um mecanismo para suporte à metaprogramação extensível para a linguagem *XAJ*. Assim, neste capítulo, são discutidas, em mais detalhes as técnicas e ferramentas diretamente relacionadas ao trabalho. Uma vez que *XAJ* permite definir extensões para a linguagem *AspectJ*, inicialmente, é apresentada uma breve explicação sobre a técnica de programação orientada a aspectos, incluindo-se uma visão superficial de *AspectJ*, que é a principal linguagem orientada a aspectos. Em seguida, é apresentada a técnica de metaprogramação, mostrando algumas linguagens e ferramentas que a utilizam, fazendo uma breve comparação com o trabalho desenvolvido. A linguagem *Meta-AspectJ* é explicada em uma seção à parte, por ser muito importante ao trabalho. Por fim, é apresentada a linguagem XAJ, para a qual o mecanismo foi desenvolvido.

2.1 Programação orientada a aspectos

A programação orientada a aspectos (*AOP*, do inglês *Aspect Oriented Programming*) (Kiczales et al., 1997) surgiu em 1997 como um novo paradigma de programação, visando resolver problemas de modularidade que ocorriam com a utilização da programação orientada a objetos (*OOP*, do inglês *Object Oriented Programming*). Em *OOP*, notou-se que alguns requisitos do programa ficavam espalhados pelo seu código, como exemplo o registro de operações e o controle de autorização. Assim foram detectados dois problemas principais de modularidade. Um problema é o *espalhamento* (*scattering*, em inglês), onde o código de uma funcionalidade ou requisito se espalha por diversos módulos. O outro problema é a *intrusão* (*tangling*, em inglês), onde o código de um ou mais requisitos aparecem em outro módulo.

Essas funcionalidades ou requisitos do programa que causam esses problemas

ficaram conhecidos como *requisitos transversais* (*crosscutting requirements*, em inglês). Na *AOP*, buscou-se uma forma de encapsular tais requisitos, eliminando os problemas de espalhamento e intrusão, aumentando, assim, a modularidade. Para isso, criou-se o conceito de *aspecto* (*aspect*, em inglês), que seria uma nova unidade de compilação que encapsula os requisitos transversais.

O aspecto encapsula o código de um requisito transversal, mas é preciso também descrever como esse código interage com o código de outros módulos, inclusive outros aspectos. Assim, surgiu o conceito de *pontos de junção* (*joinpoints*, em inglês), que são todos os pontos de um programa que podem ser alterados por um ou mais aspectos. Desse modo, além de encapsular o código do requisito, o aspecto descreve quais pontos de junção vão ser alterados por ele. Ao conjunto de pontos a serem alterados chamamos de *pontos de atuação* (*pointcuts*, em inglês). O código a ser inserido em um *pointcut* é chamado *adendo* (*advice*, em inglês).

O compilador de uma linguagem orientada a aspectos deve fazer a ligação entre o código do programa e dos aspectos. A essa nova atribuição do compilador deu-se o nome de *costura* (*weaving*, em inglês), pois o que se faz é uma costura entre os códigos.

Pode-se, portanto, encapsular os requisitos transversais de uma forma melhor, aumentando a modularidade do programa. Com isso, tende-se a aumentar a reusabilidade do código. Outra vantagem da utilização da *AOP* é um aumento na escalabilidade do código, pois pode-se acrescentar código ao programa, sem ter de inserir implentações de requisitos transversais no novo código, bastando fazer pequenas alterações nos aspectos que ali atuam, se necessário.

2.1.1 AspectJ

A principal linguagem com conceitos de *AOP* é *AspectJ* (Laddad, 2003). *AspectJ* é uma extensão da linguagem *Java* para suportar os conceitos de *AOP*, idealizada por Gregor Kiczales. Os programas em *AspectJ* são constituídos de classes *Java* e aspectos, cuja declaração se assemelha bastante à declaração de uma classe *Java*. Não é necessária a presença de aspectos para ser um programa em *AspectJ*, pois, como a linguagem é uma extensão de *Java*, qualquer programa em *Java* pode ser compilado por um compilador de *AspectJ*.

Os pontos de junção que podem ser alterados com a linguagem podem ser a execução ou chamada de métodos, leitura ou escrita de atributos e execução de inicialização estática, dentre outros. Os adendos podem vir antes, depois, ou em volta (antes e depois) de um pointcut.

A transversalidade em *AspectJ* pode ser dinâmica ou estática. A transversali-

dade dinâmica permite alterar o comportamento do programa, um exemplo pode ser a alteração feita em um dos pontos de junção citados anteriormente. A transversalidade estática permite alterar a estrutura estática do programa, um exemplo pode ser uma introdução de um atributo ou método em uma classe.

Para um exemplo de um programa em *AspectJ*, é apresentado um sistema de registro de operações financeiras simplificado. Para isso, tome por base a classe da Figura 2.1.

```
1 public class Conta{
2    ...
3    public Conta(...){...}
4    public float getSaldo(){...}
5    public void creditar(float d){...}
6    public void debitar(float d){...}
7    ...
8 }
```

Figura 2.1. Classe de representação de conta para exemplo de AspectJ.

A Figura 2.1 representa uma classe em *Java* com uma versão bem simplificada de uma conta bancária. Como uma aplicação bancária deve ser uma aplicação segura, deve-se fazer um registro de operações para um controle da movimentação de dinheiro na conta. Para isso, é apresentado, na Figura 2.2, um aspecto para um registro de operação bem simples.

```
1  public aspect Registro{
2     public pointcut operacoes():
3        execution(public void Conta.creditar(float))
4        && execution(public void Conta.debitar(float));
5     before(): operacoes(){
6        System.out.println("Entrando␣no␣método:␣"
7              + thisJoinPoint.getSignature());
8     }
9     after(): operacoes(){
10       System.out.println("Saindo␣do␣método:␣"
11             + thisJoinPoint.getSignature());
12    }
13 }
```

Figura 2.2. Aspecto para registro de operações.

Como se pode ver na Figura 2.2, a declaração de aspecto é similar à declaração de uma classe. Na linha 2, pode-se observar a declaração de um pointcut chamado de *operacoes*. O pointcut *operacoes* é constituído da execução dos métodos públicos *creditar* e *debitar*, ambos da classe *Conta* com um argumento do tipo *float*. O adendo da linha 5 imprime uma mensagem com a assinatura do ponto de junção em que está

atuando, antes da execução do mesmo. O adendo da linha 9 faz a mesma coisa depois da execução do ponto de junção.

A linguagem *XAJ*, usada neste trabalho, é uma extensão de *AspectJ*, que permite estender sua própria sintaxe concreta. A semântica de um novo comando em *XAJ* é definida pela sua tradução para *AspectJ* puro, usando metaprogramação. As seções seguintes apresentam informações sobre metaprogramação e sobre as linguagens *Meta-AspectJ* e *XAJ*.

2.2 Metaprogramação

Metaprogramação, ou *programação gerativa*, é uma técnica que consiste em escrever programas que geram ou manipulam programas. *Metalinguagem* é a linguagem que permite o uso de tal técnica. A linguagem dos programas gerados ou manipulados é chamada de *linguagem objeto*. Algumas linguagens possuem a habilidade de funcionar como sua própria metalinguagem, ou seja, a metalinguagem é igual à linguagem objeto. A essa habilidade denomina-se *reflexão* (Sobel & Friedman, 1996).

Com a metaprogramação, é possível uma redução no número de linhas na implementação de uma solução. Assim, é possível uma redução no tempo de desenvolvimento, aumentando a produtividade. A técnica permite também ao programador usufruir de maior flexibilidade, gerenciando novas situações sem necessidade de recompilação. A metaprogramação é utilizada em muitas *DSLs* e ferramentas de automação de *software* (Cordy et al., 1991).

A metaprogramação pode ser estruturada ou não estruturada. Na metaprogramação estruturada, o código a ser gerado é verificado na compilação, garantindo uma estrutura sintaticamente correta (Zook et al., 2004). Sistemas de macros usam metaprogramação para definir a tradução de novas construções em linguagens como *Scheme* (Dybvig, 2009), *Template Haskell* (Sheard & Jones, 2002), *Fortress* (Steele, 1998) e mesmo pelo pré-processador da linguagem *C++*, de forma mais simples. Em Visser (2002), os autores destacam as vantagens de se ter uma aproximação entre a notação para a manipulação abstrata dos programas objeto e a sintaxe concreta usada para os programas, aplicando a linguagem de definição de sintaxe *SDF* (Heering et al., 1989) para criar metalinguagens arbitrárias.

O mecanismo de metaprogramação utilizada pela linguagem *Fortress* possui características bem parecidas com o trabalho aqui descrito, sendo também utilizado para a tradução de novas construções. Para melhor visualização, é exibido um exemplo na Figura 2.3.

```
grammar ForLoop extends {Expression, Identifier}
  Expr | :=
    for {i:Id ← e:Expr, ?Space}* do block:Expr end ⇒
    < [ for2 i**; e** do block; end ] >
  | for2 i:Id*; e:Expr*; do block:Expr; end ⇒
    case i of
      Empty ⇒
        < [ block ] >
      Cons(ia, ib) ⇒
        case e of
          Empty < [ throw Unreachable ] >
          Cons(ea, eb) ⇒
            < [ ((ea).loop(fn ia ⇒ (for2 ib**; eb**; do block; end ))) ] >
      end
    end
end
...
// Using the new construct
g1 = < 1, 2, 3, 4, 5 >
g2 = < 6, 7, 8, 9, 10 >
for i ← g1, j ← g2 do println "(" i "," j ")" end
```

Figura 2.3. Exemplo de definição do loop *for* em Fortress.

Nesse exemplo mostrado na Figura 2.3, é definido um comando for. A sintaxe do novo comando é apresentada na terceira linha. O meta-símbolo "*" indica repetição, assim como em expressões regulares. Dessa forma, a sintaxe do comando é expressa pela palavra-chave "for" seguida de uma sequência de atribuições de uma expressão a um identificador, a palavra-chave "*do*", um bloco de comandos e termina com a palavra-chave "*end*". A semântica do comando é definida após o símbolo "⇒", começando na quarta linha. A parte entre "< [" e "] >" é a parte que se utiliza da metaprogramação, para gerar uma tradução para o comando através de uma *AST*. Como se pode observar, a tradução do comando utiliza o comando "for_2", que tem sua sintaxe e semântica definidas a partir da quinta linha.

Esse mecanismo de tradução é semelhante ao mecanismo proposto para a linguagem XAJ, que utiliza a metaprogramação com uma sintaxe simplificada bem próxima da sintaxe concreta da linguagem. Isso pode ser observado pela semelhança entre a semântica definida para o comando "for_2", na quinta linha, e o uso de tal comando na metaprogramação, na quarta linha. Outro ponto de semelhança com o mecanismo proposto é a utilização de um mecanismo de metaprogramação extensível, em que são utilizados comandos definidos pelo usuário, não se limitando às construções predefinidas, o que se nota na utilização do comando "for_2", definido a partir da quinta linha,

na metaprogramação.

No entanto, não há um compilador estável que implemente todas as facilidades de extensibilidade propostas pela linguagem *Fortress*. Essas facilidades apresentam-se apenas em um interpretador, implementado com a ferramenta *Rats!* (Grimm, 2006) que utiliza tecnologia de *Parsing Expression Grammars* (Ford, 2004). O projeto da linguagem foi encerrado em julho de 2012, dadas as dificuldades encontradas no sistema de tipos da linguagem (Steele, 2012).

Outra importante ferramenta que usa o conceito de metaprogramação é *MetaJ* (de Oliveira et al., 2004), um ambiente extensível para desenvolvimento de metaprogramas em *Java*. *MetaJ* reúne um conjunto de conceitos que são independentes da linguagem base: árvores de sintaxe, referências de código, iteradores e *templates*. Essa independência é propiciada por meio do isolamento de características comuns à maioria das linguagens, definindo operações genéricas para elas. Para introduzir componentes que são dependentes da linguagem alvo, é possível definir *pluggins* específicos para cada linguagem. Metaprogramas são escritos em *Java*.

MetaJ seria uma opção para definir o mecanismo extensível de metaprogramação de *XAJ*. No entanto, preferiu-se a linguagem *Meta-AspectJ*, uma vez que esta já oferece recursos para metaprogramação em *AspectJ* (que é uma extensão de *Java*), apesar de não oferecer recursos para a extensibilidade.

A linguagem *Meta-AspectJ*, como o nome sugere, é uma linguagem de metaprogramação para a linguagem *AspectJ*. Essa linguagem é abordada na seção seguinte, por ter uma ligação direta com o trabalho proposto. Assim, também, a linguagem *XAJ*, tratada na Seção 2.4.

2.3 Meta-AspectJ

A linguagem *Meta-AspectJ* (Huang & Smaragdakis, 2006), também conhecida como *MAJ*, é uma linguagem que permite gerar programas em *AspectJ*, através da metaprogramação. A metalinguagem de *MAJ* é uma extensão de *Java*. Para suportar a metaprogramação, foram adicionados dois operadores à linguagem. Os operadores são chamados de *quote* e *unquote*, representados por ` e #, respectivamente.

O operador *quote* marca o início do trecho para o qual deve ser gerada a *AST*. Para melhor delimitação, além do operador, o trecho deve estar entre colchetes. Assim, onde se encontra o operador, o trecho será substituído por uma construção de uma *AST*. Como a *AST* gerada pode ser de diferentes tipos, foi acrescentada a linguagem o tipo *infer*. Com o novo tipo, não há a necessidade de saber o tipo da *AST* resultante. O

tipo da variável do tipo *infer* será inferido em tempo de compilação.

O operador *unquote* é utilizado para representar uma metavariável. Uma metavariável é uma variável que representa uma *AST*. Assim, é possível a utilização de uma *AST* já construída na construção de uma nova *AST*. Uma metavariável representa uma parte variável dentro da metaprogramação, ou seja, dentro do operador *quote*.

A Figura 2.4 apresenta um exemplo simplificado de metaprogramação em MAJ. Nele se constrói uma *AST* para uma declaração do aspecto *Foo*. A seguir, são ressaltados exemplos dos conceitos supracitados.

```
1  public static void main(String myargs[]) {
2          infer pdec1 = `[private pointcut pc();];
3          infer adv1 = `[before():get(int Foo.y) {}];
4          infer var1 =  `[int x;];
5          infer mods1 = `[public static];
6          infer meth1 = `[int foo(int x, int y) {
7                              #mods1
8                              int z;
9                              x++;
10                             return x;
11                           }];
12         infer aspect2 = `[aspect Foo {
13                             #pdec1
14                             #adv1
15                             #pdec1
16                             #var1
17                             #meth1
18                           }];
19 }
```

Figura 2.4. Exemplo de metaprogramação em MAJ.

Na linha 2 da Figura 2.4, pode-se observar a utilização do operador *quote* para a construção de uma *AST* de declaração de um *pointcut* de nome *pc*. Nessa mesma linha, observa-se que foi declarada uma variável *pdec1* do tipo *infer* para receber a *AST* construída pelo operador *quote*. Ainda, mostra-se a utilização do operador *unquote* na linha 13, utilizando-se a metavariável *pdec1* que contém a *AST* mencionada anteriormente.

A linguagem *MAJ* não permite gerar qualquer tipo de *AST*, possuindo alguns pontos de entrada para a metaprogramação. Alguns dos pontos de entrada são modificadores, identificadores, unidades de compilação, declarações de adendos, entre outros. A linguagem oferece um método auxiliar para obter o código de uma metavariável. Assim, no exemplo apresentado, podemos inserir o seguinte comando *System.out.println(aspect2.unparse());* para obter a impressão do aspecto representado pela *AST* gerada.

O resultado da compilação de um programa em *Meta-AspectJ* é um programa em *Java*. Para exemplificar o resultado, na Figura 2.5, é apresentada uma parte do programa resultante da compilação do código apresentado na Figura 2.4. Na Figura 2.5, é apresentada apenas a parte equivalente à declaração de método, na linha 6, da Figura 2.4.

```
1  public class aspecttest {
2    public static void main(String[] myargs)  {
3      ...
4      MethodDec meth1 = new MethodDec(new Modifiers(0),
5        new ResolvedTypeD(new MajUncheckedType(new
6        Identifier("int"))), new Identifier("foo")
7        .getString(), new Formals().addChild(new
8        FormalDec(new Modifiers(0), new
9        ResolvedTypeD(new MajUncheckedType(new Identifier
10       ("int"))), new Identifier("x"))).addChild(new
11       FormalDec(new Modifiers(0), new ResolvedTypeD(new
12       MajUncheckedType(new Identifier("int"))), new
13       Identifier("y"))), null, new BlockStmt(new Stmts()
14       .addChild(new CodeBody(new Stmts().addChild(new
15       VarDec(mods1,new ResolvedTypeD(new MajUncheckedType
16       (new Identifier("int"))),new Identifier(new
17       Identifier("z")))).addChild(new ExprStmt(new
18       PostfixExpr(new VarExpr(new Identifier(new
19       Identifier("x"))), "++"))).addChild(new
20       ReturnStmt(new VarExpr(new Identifier(new
21       Identifier("x"))))).getChildren())).getChildren())
22     );
23     ...
24   }
25  }
```

Figura 2.5. Exemplo de programa gerado usando *MAJ*.

Compreende-se, pela Figura 2.5, que a construção de uma *AST* para uma declaração de método é bem complicada sem o uso da metaprogramação. Para uma construção à mão da *AST*, é necessário conhecer todas as classes e seus construtores. O uso da metaprogramação com uma sintaxe concreta simplificada facilita o trabalho, aumentando a produtividade, além de esconder detalhes da implementação do compilador, como as classes que compõem a *AST*.

Uma característica importante de *Meta-AspectJ* é que os trechos de programa escritos com metaprogramação são verificados estaticamente. Apenas trechos sintaticamente corretos podem gerar programas, ou seja, os elementos definidos entre "`[...]" não são tratados apenas como uma cadeia de caracteres que é tratada em tempo de execução. A estrutura sintática desse trecho é verificada em tempo de compilação, permitindo geração eficiente do código. Essa característica está diretamente relacionada à motivação do trabalho apresentado nesta dissertação. Uma vez que toda a estrutura é

analisada estaticamente, não é simples criar novos elementos que estendam a linguagem e que ainda sejam corretamente processados estaticamente.

O trabalho aqui proposto utiliza da linguagem *Meta-AspectJ* para facilitar a tradução de novos comandos em *XAJ*. Para melhor compreender o trabalho, a linguagem *XAJ* é explicada na seção seguinte.

2.4 eXtensible AspectJ

A linguagem *eXtensible AspectJ* (Di Iorio et al., 2009), também conhecida como *XAJ*, é uma linguagem que permite estender a sintaxe concreta de *AspectJ*. Uma das vantagens da linguagem é a sua modularidade, pois as extensões feitas pela linguagem ficam encapsuladas em uma nova unidade de compilação, chamada de *classe sintática*.

A declaração de uma classe sintática se assemellha muito à declaração de uma classe comum. A classe sintática contém as definições sintática e semântica da nova estrutura. Além disso, a classe sintática é usada para representar o nó da nova construção na *AST*.

A definição semântica de uma extensão é definida através de uma tradução para estruturas já conhecidas da linguagem, podendo conter ainda operações na *AST*. Neste estudo, sugere-se a utilização da metaprogramação para ajudar na construção da *AST*, obtendo suas vantagens como a ocultação de detalhes de implementação. A definição semântica deve ser definida dentro de um método especial chamado *"desugar"*.

Na Figura 2.6, é apresentado um exemplo de extensão definida em *XAJ*, que define o comando *fae*, apresentado no Capítulo 1.

```
1  public syntaxclass Fae {
2    @grammar extends statement {
3      Fae ::= ``fae'' t=type i=IDENT ``from''
4                        e1=expression ``where''
5                        e2=expression b1=compoundStatement
6                        (``else'' b2=compoundStatement)?;
7    }
8    public AST desugar(Context ctx) {
9      return`[{#t #i;
10               boolean done = false;
11               for(Iterator it = #e1.iterator(); it.hasNext();) {
12                 #i = it.next();
13                 if(#e2) {
14                   done = true; #b1
15                 }
16               }
17               if(!done) #b2}];}
18 }
```

Figura 2.6. Exemplo de classe sintática de *XAJ*.

Pode-se observar, na Figura 2.6, que a declaração de uma classe sintática é idêntica à de uma classe, apenas troca-se a palavra-chave *class* por *syntaxclass*. Os símbolos *statement*, *type*, *IDENT*, *expression* e *compoundStatement* são não terminais de uma gramática de *AspectJ*. Podemos observar duas seções na classe sintática, a primeira começa com a palavra-chave *@grammar*, e a segunda é o método *desugar*.

Na seção do *@grammar*, é definida a sintaxe da nova extensão. Podemos reparar na declaração da seção uma cláusula *extends*. Essa cláusula indica qual não terminal da gramática deve ser estendido com o novo não terminal. Nesse caso, o não terminal *statement* será estendido com o novo não terminal *Fae*, indicando que a nova construção poderá aparecer em qualquer ponto do programa onde um comando (*statement*) seja usado. Em seguida, observa-se a definição do novo não terminal. Nesse caso, o comando *fae* é definido como sendo a palavra-chave "*fae*" seguida de um tipo, um identificador, a palavra-chave "*from*", uma expressão, a palavra-chave "*where*", uma expressão e um bloco de comandos, podendo ou não ser seguido de uma cláusula *else*, que corresponde à palavra-chave "*else*", seguida de um bloco de comandos. Vale ressaltar que, antes dos não terminais, temos um sinal de igual, e antes desses sinais se encontram identificadores, como *t* e *e1*, entre outros, que são variáveis, e que recebem os valores resultantes dos não terminais.

No método *desugar*, pode-se observar o uso da metaprogramação com uma sintaxe semelhante à linguagem *Meta-AspectJ*. Nesse caso, a semântica é uma simples tradução, não contendo operações na *AST*. A tradução decorre de uma substituição do nó da estrutura pela *AST* retornada pelo método *desugar*. No caso da Figura 2.6, o nó do comando *fae* será substituído por uma *AST* que corresponde a um bloco de comandos contendo os comandos que seriam equivalentes ao seu uso, como se pode perceber pela Figura 1.2. Assim, a *AST* é gerada, utilizando-se as construções já conhecidas e os valores armazenados nas variáveis através do "casamento" da sintaxe. Portanto, para o exemplo de uso do comando, mostrado na Figura 1.1, a tradução ficaria como mostrado na Figura 2.7.

Assim, para esse exemplo, será criada uma variável *p* do tipo *Person*, e uma variável *done* do tipo *boolean*, à qual será atribuído *false*, então, o comando *for*, utilizando um *iterator* (*it*) para *people*, percorre a estrutura (*people*). Para cada iteração, é atribuído o valor da expressão *it.next()* à variável criada *p*, sendo testada a expressão $p.age < 18$. Se o resultado da expressão for verdadeiro, é atribuído *true* à variável *done* e executado o bloco que contém o comando *return p;*. Ao final das iterações, é testada a negação variável *done*; se a variável estiver com o valor *false*, será executado o bloco que contém o comando *return null;*.

Portanto, com essa tradução, consegue-se expressar exatamente a semântica pro-

```
1  {
2    Person p;
3    boolean done = false;
4    for(Iterator it = people.iterator(); it.hasNext();) {
5      p = it.next();
6      if(p.age < 18) {
7        done = true;
8        {
9          return p;
10       }
11     }
12   }
13   if(!done) {
14     return null;
15   }
16 }
```

Figura 2.7. Tradução do comando *fae* para o exemplo da Figura 1.1.

posta para o comando *fae*, apresentada no Capítulo 1, que é percorrer uma estrutura e, ao encontrar um elemento que satisfaça uma condição, executar alguns comandos, executando outros comandos, caso não encontre nenhum elemento.

Capítulo 3

Metaprogramação com extensibilidade

Neste capítulo apresenta-se um exemplo de motivação, alguns detalhes do funcionameto do compilador *MAJ*, as alterações necessárias para uma metaprogramação extensível e a explicação de como o mecanismo funciona.

3.1 Exemplo para motivação

Nesta seção, é apresentado um exemplo de construção cuja especificação pode se beneficiar de um mecanismo de metaprogramação extensível.

Suponha que se deseje definir o comando apresentado no Capítulo 1. Uma possível definição do comando *select* usando a linguagem *XAJ* é apresentada na Figura 3.1. A sintaxe do comando seria a palavra-chave "*select*" seguida de um tipo, um identificador, a palavra-chave "*from*", uma expressão, a palavra-chave "*where*" e uma expressão. A semântica do comando, como sugerido no Capítulo 1, é criar uma lista, em seguida, executar o comando *fae*, utilizando o tipo, o identificador e as expressões utilizadas no comando *select*, onde cada elemento que satisfaça a condição definida é adicionado à lista, por fim é retornada a lista. Caso nenhum elemento seja encontrado, a lista receberá o valor *null*.

Se não estiver disponível um mecanismo de metaprogramação extensível, não será possível o uso de construções definidas pelo programador na metaprogramação. Assim, o código apresentado na Figura 3.1 não seria compilado, pois, no método *desugar*, é utilizado o comando *fae* dentro da metaprogramação para a definição da semântica da nova construção.

```
1  public syntaxclass Select {
2    @grammar extends statement {
3      Select ::= ``select'' t=type i=IDENT ``from''
4                          e1=expression ``where''
5                          e2=expression;
6    }
7    public AST desugar(Context ctx) {
8      return`[{List<#t> res;
9                fae #t #i
10               from #e1
11               where #e2 {
12                 res.add(#i);
13               }
14               else {
15                 res = null;
16               }
17               return res;}];
18   }
19 }
```

Figura 3.1. Exemplo de classe sintática para definição do comando *select*.

3.2 Sistema para implementação de metaprogramação extensível

Para permitir o uso de metaprogramação que inclua construções definidas pelo próprio usuário, e assim permitir definições como a sugerida na Seção 3.1, desenvolvemos uma integração da linguagem *Meta-AspectJ* com *XAJ*. A semântica dos comandos em XAJ é especificada em *MAJ*, com os benefícios de facilidade e clareza proporcionados pela metaprogramação. Como *MAJ* só permite na metaprogramação um conjunto de construções predefinidas, implementamos um mecanismo de extensão que utiliza as definições em *XAJ* para modificar o funcionamento do *MAJ*.

Para descrever melhor o projeto, inicialmente, explica-se o funcionamento do compilador *MAJ*. Para melhor visualização, é apresentado um diagrama do compilador na Figura 3.2.

O compilador *MAJ* é composto de quatro gramáticas e um mecanismo de impressão. As gramáticas são escritas usando a linguagem da ferramenta ANTLR (Parr & Quong, 1995), versão 2. A gramática de entrada é uma gramática de análise léxica, chamada de "*Meta.g*". Os desenvolvedores optaram por implementar uma gramática de análise léxica para que os códigos dos *tokens*, que são os itens léxicos reconhecidos pela análise, fossem aproveitados por todas as outras gramáticas.

Em seguida, encontram-se as duas principais gramáticas do compilador. Os *tokens* reconhecidos pela análise léxica são passados para a gramática chamada "*MetaAspectJ.g*". Essa gramática é responsável por analisar a metalinguagem, ou seja, a

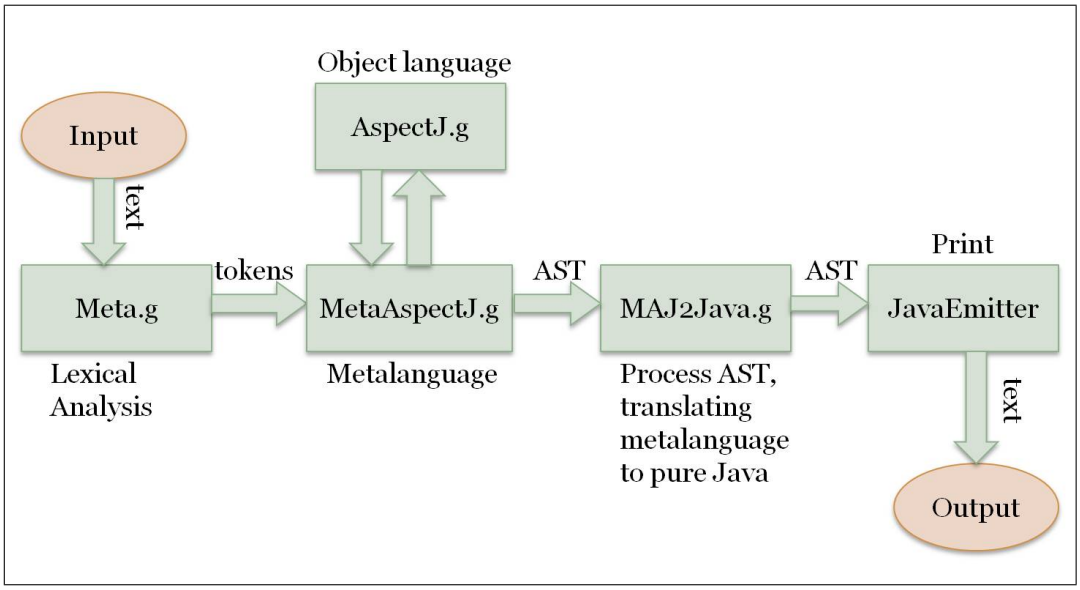

Figura 3.2. Esquema de funcionamento do compilador MAJ.

linguagem que permite o uso da metaprogramação. Quando encontra um trecho de código com metaprogramação, a gramática passa a entrada para a gramática chamada de "*AspectJ.g*". Essa gramática é responsável pela linguagem objeto, ou seja, a linguagem para a qual se deseja gerar um programa.

A árvore gerada por esse processo é passada para uma espécie de gramática chamada de "*MAJ2Java.g*". Não se trata de uma gramática normal, que analisa a sintaxe de um arquivo de entrada. A ferramenta *ANTLR*, a partir da sua segunda versão, permite a definição de gramáticas que analisam uma *árvore de sintaxe abstrata* (*AST*), uma estrutra de duas dimensões, e permite também associar ações semânticas às produções. Essas gramáticas são chamadas de *tree grammars*. A gramática *MAJ2Java.g* analisa a árvore gerada, transformando o trecho que utiliza metaprogramação, substituindo-o pelos comandos de criação dos nós da AST. Em seguida, a AST modificada é passada para o mecanismo de impressão, chamado de "*JavaEmitter*". O mecanismo é o responsável pela impressão do programa resultante, para isso ele percorre a AST, imprimindo o código correspondente a cada nó.

Para obter um mecanismo de metaprogramação extensível, é necessário alterar todas as gramáticas do compilador, além do mecanismo de impressão.

Na Figura 3.3, é apresentado um esquema do funcionamento do sistema desenvolvido neste trabalho, que usa uma especificação *XAJ*, para construir alterações nas gramáticas do *MAJ*. Usando as extensões definidas por classes sintáticas, não apenas a sintaxe da linguagem base é estendida. As novas construções são incorporadas às

gramáticas do *MAJ*, que são então recompiladas, gerando uma nova versão para a ferramenta. Essa nova versão é utilizada para processar a linguagem estendida, podendo conter instâncias das novas construções dentro dos trechos de metaprogramação.

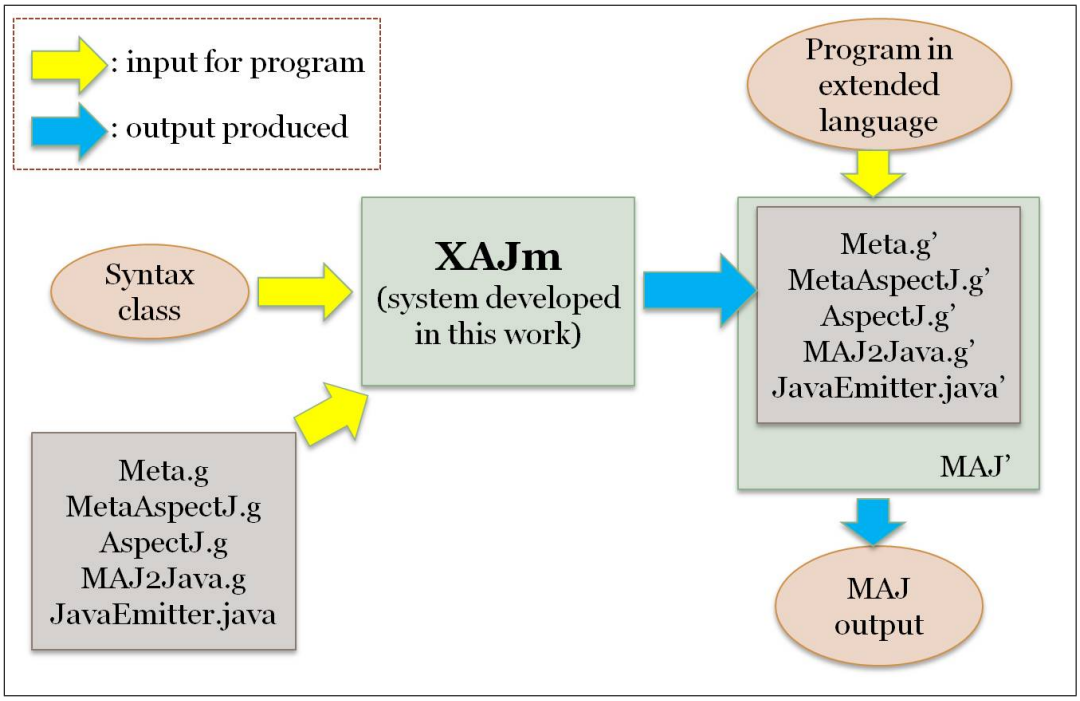

Figura 3.3. Representação do funcionamento do sistema.

3.3 Alterações processadas em *MAJ*

Para melhor compreensão do trabalho de extensão das gramáticas de *MAJ*, são apresentadas as mudanças necessárias para implementar o exemplo do comando *fae*, que foi apresentado na Figura 2.6. Após proceder às modificações descritas a seguir, é possível usar o comando *fae* para definir outros comandos (como o *select*), usando metaprogramação.

Na Figura 3.4, são apresentadas as alterações que devem ser feitas na gramática *MetaAspectJ.g*. Mostra-se que o não terminal *statement* foi estendido com o novo não terminal *Fae*, como foi definido na cláusula *extends* da declaração da classe sintática na Figura 2.6. Outra mudança na gramática foi a adição da definição do não terminal *Fae*. Podem ser notadas algumas diferenças, na definição sintática, apresentadas nas Figuras 2.6 e 3.4. Essas mudanças são adaptações da sintaxe de *XAJ* à ferramenta utilizada pelo compilador *MAJ* para a construção de gramáticas, que é a ferramenta *ANTLR*, na versão 2. Assim, adaptações serão necessárias em todas as outras gramáticas.

```
1  statement :
2        ...
3        |Fae;
4    ...
5  Fae :
6    "fae"! type IDENT "from"! expression[null]
7    "where"! expression[null] compoundStatement[null]
8    ("else"! compoundStatement[null])?
9    { #Fae =
10      #([Fae, "FAE"], #Fae);};
```

Figura 3.4. Código a ser inserido na gramática *MetaAspectJ.g*.

Outra alteração que pode ser notada é a adição de uma ação semântica, ao final da definição do não terminal *Fae*. Essa ação foi adicionada para facilitar o processo de impressão do programa resultante. A ação semântica apenas cria um nó pai do tipo *Fae* contendo um texto ("FAE"), e armazena as informações que descrevem um nó representando o comando *fae* na árvore de sintaxe abstrata. Nem todos os elementos sintáticos fazem parte desse nó da *AST*. Pode-se observar que os elementos que são constantes textuais, como *"fae"*, *"from"*, *"where"* e *"else"*, são sistematicamente seguidos de um símbolo "!". Esse símbolo faz parte do conjunto de operadores de *ANTLR* para a construção automática de árvores de sintaxe abstrata, e indica que os elementos marcados não farão parte da *AST* construída, simplificando o formato e mantendo na árvore apenas as informações realmente necessárias. O símbolo "#" é outro operador de *AST*, e serve para indicar a construção de um novo nó e a definição do seu tipo.

Na Figura 3.5, são apresentadas as alterações que devem ser feitas na gramática *AspectJ.g*. Novamente, os elementos que são constantes textuais são sistematicamente seguidos de um símbolo "!", indicando que não farão parte da *AST* construída. Podemos notar que os nomes dos símbolos não terminais são diferentes dos apresentados na Figura 3.4. Assim, temos de identificar equivalências entre os não terminais. Na Tabela 3.1, são apresentadas algumas equivalências dos não terminais das gramáticas *MetaAspectJ.g* e *AspectJ.g*, além dos não terminais utilizados para o exemplo. Para realizar essa correspondência entre os símbolos, o sistema armazena uma tabela de equivalências que foi construída após uma análise comparativa entre as gramáticas. Assim, as alterações apresentadas na Figura 3.5 são equivalentes às feitas na gramática *MetaAspectJ.g*, apresentadas na Figura 3.4.

É necessário também modificar o analisador da árvore (gerado a partir da *tree grammar MAJ2Java.g*), pois é necessário atualizar o modo de percorrer a árvore. Esse analisador irá visitar os elementos da árvore de sintaxe abstrata construída pelas gramáticas *MetaAspectJ.g* e *Aspect.g*. Como novos nós foram acrescentados, dada a definição

Não terminais	
MetaAspectJ.g	**AspectJ.g**
statement	aspectJbasicStatement
type	aspectJtypeSpec
IDENT	aspectJIDENT
expression	aspectJexpression
compoundStatement	aspectJcompoundStatement
compilationUnit	aspectJcompilationUnit
packageDefinition	aspectJpackageDefinition
importDefinition	aspectJimportDefinition

Tabela 3.1. Tabela de equivalência de não terminais.

```
1  aspectJbasicStatement :
2        ...
3        |Fae;
4    ...
5  Fae :
6    "fae"! aspectJtypeSpec aspectJIDENT "from"!
7    aspectJexpression "where"! aspectJexpression
8    aspectJcompoundStatement ("else"!
9    aspectJcompoundStatement)?
10          { #Fae =
11              #([Fae, "FAE"], #Fae);};
```

Figura 3.5. Código a ser inserido na gramática *AspectJ.g*.

do comando *fae*, a gramática *MAJ2Java.g* deve estar preparada para reconhecer esses novos nós, nos pontos em que podem ocorrer, por meio de novas produções que serão inseridas. O analisador da árvore nesse caso apenas percorre a árvore, substituindo a parte contendo a metaprogramação pelo código de criação dos objetos. É importante ressaltar que o analisador percorre a árvore como um todo, ou seja, ela contém produções que representam as duas gramáticas (*MetaAspectJ.g* e *AspectJ.g*). Assim, é necessário estender o analisador em duas partes diferentes. As alterações necessárias são apresentadas na Figura 3.6.

O analisador da árvore apresentado funciona como um *Visitor* (Gamma et al., 1995) percorrendo a árvore. O uso do operador "#", nas gramáticas de *AST* de *ANTLR*, tem uma função complementar ao seu uso em gramáticas de análise sintática. Enquanto nas últimas o operador define a construção de um novo nó, nas primeiras, indica um nó construído entre "#(...)", sendo que o primeiro elemento entre parênteses representa o tipo do nó. Por exemplo, na linha 6 da Figura 3.6, a produção define que um nó de *AST*, com tipo *Fae* terá associado a ele informações que são outras sub-árvores, derivadas dos símbolos *type*, *IDENT*, *expression* e *slist*. Pode-se observar a correspondência

```
1  stat :
2        ...
3        |Fae;
4    ...
5  Fae :
6    #(Fae type IDENT expression
7         expression slist (slist)?);
8    ...
9  aspectJstat :
10       ...
11       |aspectFae;
12   ...
13 aspectFae :
14   #(Fae t:aspectJtypeSpec i:aspectJIDENT
15     e1:aspectJexpr e2:aspectJexpr b1:aspectJslist
16     (b2:aspectJslist)?)
17     {if(#b2 == null){
18       #b2 = #[IDENT, "null"];}
19      #Select = #(#[EXPR, "EXPR"],
20        #(#[LITERAL_new, "new"], #[IDENT, "Fae"],
21          #(#[ELIST, "ELIST"],#t, #i, #e1, #e2,
22            #b1, #b2)));};
```

Figura 3.6. Código a ser inserido no analisador *MAJ2Java.g*.

imediata com a construção realizada na Figura 3.4, onde as constantes textuais foram descartadas. As duas expressões derivadas do não terminal *expression*, na Figura 3.6, são resultado das expressões após "from" e "where", da Figura 3.4. As duas ocorrências de *slist*, na Figura 3.6, são resultado dos dois usos de *compoundStatement* na Figura 3.4, e em ambas as gramáticas, a segunda ocorrência é opcional.

Mais uma vez, comparando-se as gramáticas das Figuras 3.4, 3.5 e a *tree grammar* da Figura 3.6, podemos perceber que o nome de alguns não terminais são diferentes, sendo necessário, assim, fazer as devidas equivalências. Essas equivalências também se encontram na tabela armazenada pelo sistema. A Tabela 3.2 mostra as equivalências com os não terminais apresentados na Tabela 3.1. Pode-se perceber, nas duas últimas produções, não terminais com o prefixo *aspect*. Esse prefixo serve para diferenciar as produções da linguagem objeto. Na produção do não terminal *Fae*, o analisador apenas percorre o comando, pois esse seria o comando da metalinguagem. Na produção do não terminal *aspectFae*, o analisador o substitui pela expressão com a chamada do construtor da classe correspondente, pois esse é o comando da linguagem objeto.

A ação semântica para substituir o nó pela chamada do construtor segue o modelo apresentado na Figura 3.7. Podemos observar no exemplo da Figura 3.6 um comando *if* inicial. Esse comando serve para tratar a parte opcional do comando.

Além de todas as alterações descritas, é necessário estender o analisador léxico da gramática *Meta.g*, acrescentado as novas palavras-chave definidas que, nesse caso,

		MAJ2Java.g
MetaAspectJ.g	statement	stat
	type	type
	IDENT	IDENT
	expression	expression
	compoundStatement	slist
	compilationUnit	compilationUnit
	packageDefinition	packageDefinition
	importDefinition	importDefinition
AspectJ.g	aspectJbasicStatement	aspectJstat
	aspectJtypeSpec	aspectJtypeSpec
	aspectJIDENT	aspectJIDENT
	aspectJexpression	aspectJexpr
	aspectJcompoundStatement	aspectJslist
	aspectJcompilationUnit	aspectJcompUnit
	aspectJpackageDefinition	aspectJpackageDefinition
	aspectJimportDefinition	aspectJimport

Tabela 3.2. Tabela de equivalência de não terminais.

```
1 {#<Nome do não terminal> = #(#[EXPR, "EXPR"],
2    #(#[LITERAL_new, "new"], #[IDENT, "<Nome␣da␣classe>"],
3      #(#[ELIST, "ELIST"],
4        <Parâmetros>)));};
5 }
```

Figura 3.7. Padrão para a chamada dos construtores.

seriam: "*fae*", "*from*" e "*where*". Finalmente, é necessário estender, também, o mecanismo para imprimir a nova estrutura sintática. As alterações para o mecanismo de impressão são apresentadas na Figura 3.8.

O mecanismo é implementado como uma função recursiva, utilizando um comando *switch*. Como foram adicionadas aquelas ações semânticas nas Figuras 3.4 e 3.5, basta acrescentar um novo *case*.

A AST também deve ser estendida, criando uma classe para representação da nova estrutura. Lembrando que em *Meta-AspectJ* existe um método auxiliar para obter o código de uma metavariável, é necessário sobrescrever o método *unparse* em todas as novas classes criadas. Esse método apenas retorna o texto contendo o código da metavariável. Sua implementação é bem semelhante à implementação do mecanismo citado anteriormente, mudando apenas alguns nomes de funções utilizadas.

```
1  ...
2  case Fae:
3    out.print("fae␣");
4    AST aux = ast.getFirstChild();
5    print(aux);
6    out.print("␣");
7    aux = aux.getNextSibling();
8    print(aux);
9    out.print("␣from␣");
10   aux = aux.getNextSibling();
11   print(aux);
12   out.print("␣where␣");
13   aux = aux.getNextSibling();
14   print(aux);
15   out.print("␣");
16   aux = aux.getNextSibling();
17   print(aux);
18   aux = aux.getNextSibling();
19   if(aux != null){
20     out.print("␣else␣");
21     print(aux);}
22   break;
23  ...
```

Figura 3.8. Código a ser inserido no *JavaEmitter*.

3.4 Automatização do processo

Na Seção 3.3, foram apresentadas as várias alterações que devem ser realizadas nas gramáticas do *MAJ*, para implementar o exemplo de definição do comando *fae*. Nesta seção, descrevemos como o sistema implementa essas alterações, e as limitações atuais.

A linguagem *XAJ* foi construída como uma extensão do compilador *abc* (Avgustinov et al., 2005) que, por sua vez, usa a ferramenta *Polyglot* (Nystrom et al., 2003). Assim, foi necessário modificar o compilador *MAJ*, para que sua metaprogramação gerasse as classes de representação da *AST* utilizadas pelo compilador *abc*. Desse modo, o programa resultante poderá ser compilado pelo compilador de *XAJ*.

Além disso, um programa escrito em *XAJ*, ao fazer referências aos não terminais, utiliza os não terminais da gramática utilizada pelo compilador *abc*, pois são essas as gramáticas utilizadas por seu compilador. Assim, foi necessário fazer as equivalências entre os não terminais das gramáticas utilizadas pelo compilador *MAJ* e o *abc*. Logo, além das tabelas de equivalências entre os não terminais das gramáticas de *MAJ*, é armazenada no sistema uma tabela de equivalência entre os não terminais das gramáticas do compilador *abc* e *MAJ*.

Como foi apresentado na Seção 3.2, as gramáticas do compilador *MAJ* foram escritas, utilizando a ferramenta *ANTLR*, na versão 2. Essa versão da ferramenta não dá suporte à extensão de gramática, o que dificulta um pouco o processo. Assim, o

mecanismo proposto não deve gerar apenas extensões para as gramáticas de *MAJ*, mas as gramáticas completas, contendo suas respectivas alterações.

As alterações consistem, basicamente, em adicionar a definição sintática contida na seção *@grammar*, fazendo-se as adaptações para a ferramenta *ANTLR*, e estendendo-se o não terminal contido na cláusula *extends* da mesma seção com um não terminal com o mesmo nome da classe sintática. É importante lembrar que essas alterações devem ser feitas em todas as gramáticas citadas na Seção 3.3, fazendo as equivalências de seus não terminais. Vale ressaltar que não é necessário adicionar as ações semânticas definidas na classe sintática, pois o mecanismo não é responsável pela execução do código.

Para fazer tais alterações, o mecanismo processa a classe sintática para a qual devem ser geradas as extensões. Depois de processada a entrada, o mecanismo reconhece quais os não terminais devem ser estendidos com as alterações descritas anteriormente. Então, para cada uma das gramáticas, o mecanismo deve ler a gramática, procurando os não terminais a serem estendidos, ao achar um não terminal sua produção é copiada e, ao final, é acrescida uma nova alternativa contendo apenas o novo não terminal.

O mecanismo de impressão deve imprimir o código da nova estrutura, mas uma boa formatação é uma questão de estilo e gosto pessoal (Leavens, 1984). Assim, o código para esse mecanismo de impressão (*JavaEmitter*) não é gerado automaticamente, pois não há como reconhecer a formatação desejada pelo programador, utilizando-se apenas as informações contidas na classe sintática na sua versão atual. Assim, o código apresentado na Figura 3.8 foi feito a mão.

Depois de processado o programa, o resultado será um programa em *XAJ* sem o uso da metaprogramação com sintaxe próxima à sintaxe concreta da linguagem. Assim, o código gerado pode ser compilado sem problemas, pelo compilador de *XAJ*.

Para gerar e reconhecer a linguagem *XAJ*, a definição de classes sintáticas foi adicionada à metalinguagem (*MetaAspectJ.g*). Uma vez que uma alteração em uma gramática altera a *AST* gerada, foi necessário também alterar o analisador da árvore (*MAJ2Java.g*) e o mecanismo de impressão (*JavaEmitter.java*), para assim poder imprimir a nova construção.

3.5 Avaliação

Nesta seção, apresenta-se uma avaliação do trabalho desenvolvido, concentrando-se em uma análise dos mecanismos desenvolvidos e dos benefícios alcançados.

Os benefícios de uso de metaprogramação são bem claros e foram discutidos na

Seção 2.2. No caso da ferramenta *XAJ*, destaca-se a facilidade para definir a semântica de novos comandos, especificando-se trechos de código de maneira bem similar à sintaxe concreta da linguagem. Esses benefícios estão disponíveis aos usuários da ferramenta *XAJ*, após o desenvolvimento dos mecanismos apresentados neste trabalho. Um desenvolvedor pode especificar uma nova construção em *XAJ*, e depois usar essa construção para definir a semântica de outras, continuando a se beneficiar das facilidades da metaprogramação.

O esforço que deve ser dispendido por um programador para alcançar os benefícios ora mencionados não são maiores do que o trabalho até então necessário para definir uma nova construção para a linguagem, usando classes sintáticas de XAJ. Com este trabalho, procura-se realizar todo o processamento automaticamente, a partir da definição das classes sintáticas. Apenas uma parte da geração do código ainda não está automatizada, como descrito na Seção 3.4.

Na Tabela 3.3, são apresentados números de linhas de código alteradas para a definição do comando *fae*, usado como exemplo na Seção 3.1, de modo que esse comando possa ser usado na metaprogramação. Na referida tabela, destacam-se os arquivos alterados e mostra se o código é gerado automaticamente ou não. Pode-se observar que a grande maioria das alterações nos arquivos é realizada automaticamente.

Arquivo	**Função**	**Linhas alteradas**	**Automático**
Meta.g	Analisador léxico	7	Sim
MetaAspectJ.g	Gramática da metalinguagem	5	Sim
AspectJ.g	Gramática da linguagem objeto	6	Sim
MAJ2Java.g	Analisador da árvore	12	Sim
JavaEmitter.java	Mecanismo de impressão	22	Não
	Total	52	

Tabela 3.3. Tabela para avaliação.

Concluímos que os benefícios esperados pelo sistema foram alcançados. A parte do código que ainda não pode ser gerada automaticamente está mais diretamente ligada à impressão de novas construções, mas não afeta nem prejudica o uso dessas construções na definição de outras, aplicando metaprogramação.

Capítulo 4

Conclusões

O trabalho aqui descrito apresenta um mecanismo de metaprogramação extensível para a linguagem *XAJ*. O objetivo é oferecer ao programador uma facilidade para expressar a semântica de uma nova estrutura de forma legível e independente da implementação das classes que representam a árvore de sintaxe abstrata dos programas. O mecanismo sendo extensível permite que uma nova construção seja expressa em termos de outras definidas pelo próprio programador, aumentando a produtividade. Tendo uma metaprogramação com uma sintaxe próxima à sintaxe concreta da linguagem, há um ganho na legibilidade, facilitando o entendimento da semântica das extensões. Consideramos que o trabalho alcançou os resultados esperados, gerando, automaticamente, quase todo o código necessário para a implementação das extensões, a partir de definições escritas em classes sintáticas na linguagem *XAJ*.

4.1 Dificuldades encontradas

Algumas das maiores dificuldades na implementação desenvolvida decorreram da linguagem *Meta-AspectJ* ter sido construída usando ferramenta *ANTLR* (Parr & Quong, 1995), na versão 2. Essa versão, como foi apresentado na Seção 3.4, não dá suporte à extensão de gramáticas, assim, para acrescentar uma produção, é necessário reescrever a gramática adicionando a produção. A nova versão da ferramenta possui uma sintaxe diferente e dá suporte à extensão.

Outra dificuldade foi devido à metalinguagem de *MAJ* não ser baseada em *AspectJ*, e sim em *Java*. Apenas a linguagem objeto de *MAJ* é *AspectJ*. Assim sendo, considerando as gramáticas *AspectJ.g* (linguagem objeto) e *MetaAspectJ.g* (metalinguagem), citadas na Seção 3.2, nem todas as produções da gramática *AspectJ.g* possuem uma produção equivalente na gramática *MetaAspectJ.g*. Isso dificulta a extensão

simultânea das duas gramáticas. Além disso, a gramática *AspectJ.g* não é completa, não implementa a linguagem *AspectJ* totalmente. Ao tentarmos construir algumas extensões, nos deparamos com a falta de declarações intertipo, por exemplo.

Outra dificuldade é relacionada aos conflitos que podem surgir ao estender as gramáticas. Um exemplo dos conflitos foi ao tentarmos fazer o comando *for* para coleções, já que o compilador se baseia em *Java 4*. Ao adicionarmos as produções necessárias para o novo *for*, conflitos surgiram, dificultando a especificação do comando, pelo *ANTLR* usar analisador LR. Tais conflitos surgem da extensibilidade, em *XAJ*, também podem surgir conflitos com as extensões definidas pelo usuário. O compilador de *XAJ*, até então, não consegue tratar esses conflitos.

4.2 Trabalhos futuros

Como trabalhos futuros, pode-se citar uma possível reescrita das gramáticas usadas em Meta-AspectJ, usando a ferramenta *ANTLR* na sua versão mais recente. Isso deve diminuir os possíveis conflitos em extensões, como no caso do novo comando *for* apresentado na Seção 4.1, pois a nova versão de *ANTLR* oferece suporte para *lookahead* infinito e *backtracking* na análise sintática. Além das vantagens citadas, a nova versão dá suporte à extensão de gramáticas. Assim, o mecanismo ficaria muito mais simples, criando apenas as extensões das gramáticas, sem ter de fazer uma leitura da gramática e reconstruí-la.

Outro trabalho é completar a gramática da linguagem objeto de *Meta-AspectJ* para descrever toda a linguagem *AspectJ*. Para isso, devem ser realizadas alterações na gramática *AspectJ.g*, como a inserção de declarações intertipo. Ainda relacionado à ferramenta *MAJ*, é interessante estender a gramática da metalinguagem para basear-se em *AspectJ*, tornando-se, assim, mais próxima de *XAJ*. Para isso, devem ser realizadas alterações na gramática *MetaAspectJ.g*, como a inserção da declaração de aspectos.

Outro trabalho importante é a criação de uma notação para se especificar como uma nova estrutura será impressa, pois o código para a impressão é um dos pontos para os quais ainda não é gerado código automaticamente. O código da Figura 3.8, por exemplo, precisou ser feito à mão. Essa notação poderia ser adicionada em uma seção específica na classe sintática, tendo sintaxe próxima à sintaxe da definição da estrutura, contendo símbolos especiais para espaços obrigatórios e opcionais, além de quebras de linha e indentação. Mesmo podendo utilizar soluções com base em mecanismos como o apresentado em de Jonge (2000), a formatação depende de adaptações para satisfazer o gosto e estilo do programador.

Referências Bibliográficas

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. Em *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pp. 87–98, New York, NY, USA. ACM.

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

Cordy, J. R.; Halpern, C. D. & Promislow, E. (1991). Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16:97–107.

de Jonge, M. (2000). A pretty-printer for every occasion. Em Ferguson, I.; Gray, J. & Scott, L., editores, *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pp. 68–77. University of Wollongong, Australia.

de Oliveira, A. A.; Braga, T. H.; de Almeida Maia, M. & da Silva Bigonha, R. (2004). Metaj: An extensible environment for metaprogramming in java. *Journal of Universal Computer Science*, 10(7):872–891.

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

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

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

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. *SIGPLAN Not.*, 41(6):38–51.

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. Em *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pp. 865–868, New York, NY, USA. ACM.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M. & Irwin, J. (1997). Aspect-oriented programming. Em *ECOOP*, pp. 220–242.

Laddad, R. (2003). *AspectJ in Action: Pratical Aspect-Oriented Programming*. Manning Publications Company.

Leavens, G. T. (1984). Prettyprinting styles for various languages. *SIGPLAN Not.*, 19(2):75–79.

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. Em *12th International Conference on Compiler Construction*, pp. 138–152. Springer-Verlag.

Parr, T. J. & Quong, R. W. (1995). Antlr: A predicated-ll(k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

Reis, L. V. d. S. (2010). Especialização de Linguagens Orientadas a Aspectos Baseada em Extensibilidade de Gramáticas. Master's thesis, Universidade Federal de Minas Gerais.

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

Sobel, J. M. & Friedman, D. P. (1996). An introduction to reflection-oriented programming. Em *Proceedings of Reflection'96*.

Steele, G. (2012). Fortress wrapping up, https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up.

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

van Deursen, A.; Klint, P. & Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

Visser, E. (2002). Meta-programming with concrete object syntax. Em Batory, D.; Consel, C. & Taha, W., editores, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pp. 299–315, Pittsburgh, PA, USA. Springer-Verlag.

Zook, D.; Huang, S. S. & Smaragdakis, Y. (2004). Generating aspectj programs with meta-aspectj. Em *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, volume 3286 of LNCS*, pp. 1–19. Springer.