Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

GERAÇÃO DE SISTEMAS DE TRANSFORMAÇÃO

Adeline de Sousa Silva

DISSERTAÇÃO DE MESTRADO

Recife
31 de agosto de 2006

Universidade Federal de Pernambuco
Centro de Informática

Adeline de Sousa Silva

GERAÇÃO DE SISTEMAS DE TRANSFORMAÇÃO

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Prof. Paulo Henrique Monteiro Borba*

Recife
31 de agosto de 2006

Para minha mãe e irmãs e meu cãozinho Vinny. E para aqueles que mesmo não estando mais aqui, continuam em meu coração: painho e Chispita.

AGRADECIMENTOS

Adeline estuda, estuda que a vida não tá fácil não.

—CASTOR

À minha família, que, mesmo de longe, sempre apoiou todos os meus passos. E certamente é a grande responsável por eu ter conseguido chegar até aqui.

A Paulo Borba, misto de orientador, pai, psicólogo... Sou eternamente grata a ele, por ter aguentado minhas várias crises existenciais ao longo deste mestrado e por ter me ajudado a superá-las! Nestes seis anos de convivência, divergências e muitas implicâncias, aprendi muito com ele! Quero também aproveitar para pedir desculpas pelos vários cabelos brancos que eu lhe proporcionei. :)

A André Santos, professor que sempre esteve presente na minha vida acadêmica e profissional. Agradeço a confiança que sempre depositou em mim e pela amizade sempre demonstrada nestes muitos anos de convivência.

A todos os meus professores, desde Tia Fátima que me ensinou a ler, passando por todos os professores que sempre me estimularam a estudar, até os meus professores de faculdade. Sem eles eu não estaria aqui.

Ao meu companheiro de luta Gustavo, com quem eu dividi muitas agruras da vida com o JaTS :) Foram muitos desafios, mas a gente até que conseguiu se sair direitinho.

Ao meu chefe Orlando, que me liberou por alguns dias para que eu tivesse o tempo necessário para concluir esta dissertação. E que me deu um grande apoio nesta fase final.

Ao meu grande amigo Fernando Castor, com quem isso tudo começou, 6 anos atrás.

Ao pessoal da Qualiti, pelo apoio e amizade de sempre. Em especial, aos envolvidos com o Qualiti Coder. Muito da evolução do JaTS e do C#TS foi auxiliada por vocês.

Aos membros do *Software Productivity Group*, pelas valiosas contribuições a este trabalho. Em especial a Vander Alves, Tiago Massoni, Rohit Gheyi e Sérgio Soares.

Às minhas companheiras de mestrado: Talita, Carla e Manuela. Foi muito bom poder dividir as angústias e alegrias deste período com vocês.

Ao meu amigo Deco, com quem paguei algumas cadeiras desse mestrado, que proporcionaram ótimas experiências. Até no Japão fomos parar!

Por fim, agradeço a todos os que me ajudaram neste trabalho. Seja através de discussões técnicas, de trocas de experiências ou, simplesmente, mas não menos importante, pelo apoio e amizade.

RESUMO

À medida que os sistemas se tornam mais complexos, cresce a necessidade de desenvolver componentes em mais de uma linguagem. O desenvolvimento de um sistema Web, por exemplo, envolve pelo menos três linguagens: a linguagem de desenvolvimento do núcleo do sistema (Java), a linguagem de apresentação (JSP+ HTML) e a linguagem de configuração (XML), para citar um exemplo.

Portanto, cresce também a necessidade de ferramentas que gerem e mantenham códigos escritos em várias linguagens. Sem isso, a produtividade dos desenvolvedores pode diminuir, tornando alto o custo de fabricar e manter softwares complexos.

As ferramentas independentes de linguagem têm a vantagem de transformar vários tipos de linguagens. Mas por serem puramente sintáticas, não conseguem exprimir certas transformações e também tornam mais difícil exprimir transformações complexas. As ferramentas de transformação específicas para uma determinada linguagem, por outro lado, podem realizar transformações baseadas em semântica e, com maior concisão, expressar transformações elaboradas.

Assim, para evoluir sistemas complexos, o ideal seria dispor de um conjunto de ferramentas de transformação específicas para cada linguagem de que o sistema faz uso. A construção de ferramentas deste tipo, entretanto, é bastante custosa.

Este trabalho apresenta uma abordagem de programação gerativa que tenta unir as vantagens dos dois tipos de ferramentas supracitadas, ao permitir a geração de sistemas de transformação específicos a um baixo custo.

A idéia é usar o JaTS (*Java Transformation System*) como uma arquitetura de linha de produtos de sistemas de transformação, a partir da qual nós podemos instanciar novos sistemas de transformação para outras linguagens. Para conseguir isto, nós estudamos o JaTS para definir que partes dele poderiam ser reusadas e quais deveriam ser parametrizadas. Então, nós definimos uma abordagem de programação gerativa para o processo de instanciação de novos sistemas usando transformações JaTS.

Palavras-chave: Transformação de programas, Programação Gerativa, JaTS, Linha de Produtos

ABSTRACT

As systems become more complex, the need for developing components using more than one programming language increases. For instance, the development of a web system usually involves several different languages: the core system language (Java), the presentation language(JSP+ HTML), and the configuration language (XML). Accordingly, generative and maintenance tools should be capable of dealing with this diversity of languages, transforming programs written in any language of the system's components.

Language-independent transformation systems address this issue by generically handling a number of languages, whose syntax is a parameter of the transformation engine. However, since such systems are mostly syntactic and not semantics-based, they do not support expressing more elaborate transformations for these languages. On the other hand, language-specific transformation systems support simpler specification of more elaborate transformations, since they are semantics-based.

Therefore, in order to evolve complex systems, it would be useful to rely on language-specific transformation systems for the languages employed on their development. Building such language-specific transformation systems, however, is costly.

We present in this work a method for combining the benefits of both kinds of transformation systems, thus allowing faster and less costly creation of language-specific transformation systems.

The main idea is to use the Java Transformation System (JaTS) as the architecture of a transformation system product line, from which we can instantiate specific transformation systems for other languages. In order to accomplish this, we studied which parts of JaTS could be used in a flexible transformation system architecture, and which parts could be parameterized. We further defined a generative process for the instantiation of a language-specific transformation system, by using JaTS and performing some modifications on the object-language (the language to which the transformation system is generated).

Keywords: Program Transformations, JaTS, Generative Programming, Product Lines

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTINGS

CAPÍTULO 1

INTRODUÇÃO

Toda grande caminhada começa com o primeiro passo.

—MAO TSÉ TUNG

Qualidade e produtividade, fatores chave para a competitividade, sempre foram preocupação da indústria de software, em maior ou menor escala, em diferentes setores. A busca por qualidade observou diferentes abordagens ao longo do tempo. Entretanto, com o acirramento da competição, conseqüência da globalização, a qualidade passou a ser questão de sobrevivência. Outra questão advinda da acirrada competição, é o chamado *time-to-market*: o tempo entre a idéia sair do papel e chegar ao mercado. Este tempo tem sido cada vez menor.

Portanto, a indústria de software está vivendo seu processo de aprimoramento. Saindo da fase artesanal e caminhando para uma espécie de "revolução industrial".

Nos últimos anos, o esforço para tornar o processo de desenvolvimento menos artesanal, seja através da definição de ferramentas, processos ou tecnologias cresceu bastante. Entretanto, cresceu também, em velocidade maior, a complexidade dos sistemas computacionais produzidos.

Entre os vários mecanismos, ferramentas para geração automática e reestruturação de código têm grande se mostrado bastante eficazes. A geração automática permite que componentes dos sistemas fiquem prontos mais rapidamente e a reestruturação permite realizar manutenções de forma mais fácil.

1.1 O PROBLEMA

Geração automática de código é um mecanismo bastante utilizado. De fato, a maioria dos IDEs[1] [Jet06, Ecl06, Bor06] traz suporte a geração de pequenos trechos de código e também suportam um conjunto pré-definido de reestruturações.

A questão é que à medida em que os sistemas de softwares têm se tornado mais complexos, a quantidade de linguagens envolvidas em seu desenvolvimento tende a crescer. Isto porque apenas linguagens de programação de propósito geral não são suficientes (ou

[1]Do inglês: Integrated Development Environment: Ambiente Integrado de Desenvolvimento.

não são tão produtivas) para desenvolver estes sistemas com as restrições cada vez maiores de tempo. Tomemos como exemplo um simples sistema baseado na Web: precisamos de pelo menos duas linguagens para desenvovê-lo. Uma linguagem de programação (por exemplo, Java) e outra de apresentação, por exemplo, HTML.

Então, reestruturar um sistema desses envolve reestruturar componentes em várias linguagens e, para isto, é necessário apoio de ferramentas. Já que fazer reestruturação manual é uma tarefa entediante e propensa a erros.

O ideal seria contar com ferramentas que auxiliassem o processo de construção (através de geração automática) e de manutenção (através de transformações, refatorações, etc.) e que suportassem toda a gama de linguagens usada pela aplicação. Uma alternativa para isso seria a utilização de sistemas de transformação de programas.

Para tratar a questão das várias linguagens, pode-se optar por duas abordagens: usar sistemas de transformação independentes de linguagem ou usar vários sistemas de transformação para cada linguagem envolvida nesses sistemas.

Os sistemas independentes de linguagem têm a vantagem de transformar programas escritos em várias linguagens. Porém, esta generalidade traz o inconveniente de não poder expressar transformações complexas e também de requerer que a gramática da linguagem seja entrada para a transformação. Já os sistemas específicos têm a capacidade de expressar, de forma mais simples, transformações complexas e transformações que requerem conhecimento da semântica da *linguagem objeto*.

1.2 SOLUÇÃO PROPOSTA

Em nosso trabalho, definimos uma abordagem para geração de sistemas de transformação usando técnicas de programação gerativa [CE00]. Assim, podemos unir as vantagens de ter um sistema de transformação específico para cada linguagem com a vantagem de todos os sistemas terem um único modelo de funcionamento e serem baseados em linguagens de templates com construções similares.

O JaTS [CB01, COS+01] é usado como modelo para a linha de produtos de sistema de transformação. Além disto, o JaTS é usado como ferramenta de geração de código para tais sistemas e partes da sua implementação são reusadas por estes sistemas.

Como o JaTS é usado como sistema modelo, nosso primeiro passo foi avaliar a arquitetura de JaTS e os aspectos de sua implementação a fim de identificar pontos que pudessem ser reusados e, também, identificar a estrutura das classes que seriam geradas automaticamente.

1.2.1 Contribuições

As principais contribuições deste trabalho são as seguintes:

- Identificação dos componentes de um sistema de transformação baseado em *templates* e definição de uma abordagem para geração de uma linha de produtos de sistema de transformação baseados no JaTS.
- Definição de um sistema para transformação de especificações de gramática usando o JavaCC.
- Avaliação da abordagem apresentada, através da geração de três diferentes sistemas de transformação.

1.3 ORGANIZAÇÃO DESTE TRABALHO

Apresentamos neste Capítulo 1 uma introdução ao problema tratado neste trabalho e um resumo de suas contribuições.

No Capítulo 2 apresentamos o sistema JaTS. Mostramos como é o processo de definição e aplicação de transformações e também apresentamos as construções da sua linguagem de *templates*.

No Capítulo 3 apresentamos a semântica informal das construções de JaTS-TL, pois os sistemas gerados neste trabalho possuem uma linguagem de *templates* similar a JaTS-TL, com a mesma semântica nas construções.

A abordagem para geração de sistemas de transformação proposta por este trabalho é apresentada no Capítulo 4.

O Capítulo 5 apresenta o JavaCCTS, um sistema de transformação para linguagem de especificação de gramáticas gerado usando a abordagem apresentada por este trabalho. Além de ser um estudo de caso desta abordagem, tal sistema passou a integrar as ferramentas usadas para geração de novos sistemas de transformação.

No Capítulo 6 apresentamos alguns sistemas gerados usando a abordagem apresentada neste trabalho, discutimos aspectos de suas implementações e, ao final, um resumo dos resultados alcançados.

Por fim, no Capítulo 7, apresentamos nossas conclusões, os principais trabalhos relacionados e alguns trabalhos futuros.

No Apêndice A podem ser encontradas as transformações completas usadas neste trabalho e no Apêndice B, um exemplo maior de código transformado, bem como a gramática da linguagem de especificação de gramáticas do JavaCC.

Com relação ao padrão de escrita, adotamos a seguinte convenção:

- Fontes em *itálico* são utilizadas para representar termos em língua estrangeira e/ou para dar ênfase a certas palavras e expressões.
- Trechos de código e termos dentro do texto que representem construções da linguagem serão apresentados em fonte `Courier New`.
- Trechos de código maiores (representando uma classe inteira), assim como templates serão identificados da seguinte estrutura:

Listing 1.1 Exemplo de listagem

```
class #C {}
```

CAPÍTULO 2

JATS

JaTS é um acrônimo para *Java Transformation System*, um sistema de transformação para a linguagem Java, baseado em *templates*.

Os *templates* usados por JaTS são escritos em JaTS-TL, *JaTS Template Language*, uma linguagem para especificação de *templates* de programas Java. Por ser definida como uma extensão sintática da linguagem Java, as transformações em JaTS-TL podem ser descritas de uma forma muito parecida com programas Java, facilitando o entendimento por parte de quem escreve as transformações.

Uma transformação em JaTS é composta por uma pré-condição, um conjunto de *templates* de casamento e um conjunto de *templates* de geração. Não há relação entre o número de *templates* em cada lado da transformação, porém o número de *templates* de casamento deve coincidir com o número de programas fonte Java.

Para simplificar a explicação, vamos considerar apenas transformações envolvendo um único *template* de casamento e um único *template* de geração.

A pré-condição determina se o programa deve ou não ser transformado. E é composta por uma expressão booleana, que é avaliada antes de a transformação ser aplicada. Sua definição é opcional e, caso não esteja presente, assume-se que seu valor é verdadeiro.

O *template* de casamento possui uma estrutura sintática compatível com a do programa fonte. E o *template* de geração, define a estrutura sintática do programa após a transformação.

JaTS se enquadra nos sistemas de transformação que realizam transformações horizontais. Tanto os programas de entrada (programas fonte) quanto os de saída (programas destino) são escritos na linguagem Java. Os programas fontes são utilizados para extração de informações necessárias à geração dos programas especificados pelo *template* de geração.

O processo de aplicação de uma transformação em JaTS, é realizado em três fases: *parsing*(transforma o programa texto em árvore sintática), transformação e *pretty-printing* (transforma a árvore sintática em programa texto).

A etapa de transformação é executada em três passos: (*i*) casamento de padrões, (*iii*) substituição e (*iii*) processamento.

O casamento de padrões é feito entre as árvores do *template* de casamento e do

programa fonte Java. Neste passo é gerado um conjunto de mapeamentos de variáveis presentes no *template* a valores encontrados no programa fonte Java. A este conjunto damos o nome de conjunto-resultado. Este conjunto é usado nos passos subseqüentes.

O segundo passo consiste da substituição de valores do *template* de geração. As variáveis presentes são substituídas pelos valores que a elas se encontram mapeados no conjunto-resultado.

O terceiro passo é o processamento de estruturas executáveis. Nesse passo, as declarações executáveis presentes no *template* de geração, são executadas e o resultado de seu processamento é adicionado a árvore do programa a ser gerado.

A Figura 2.1 ilustra a aplicação de uma transformação a um programa Java.

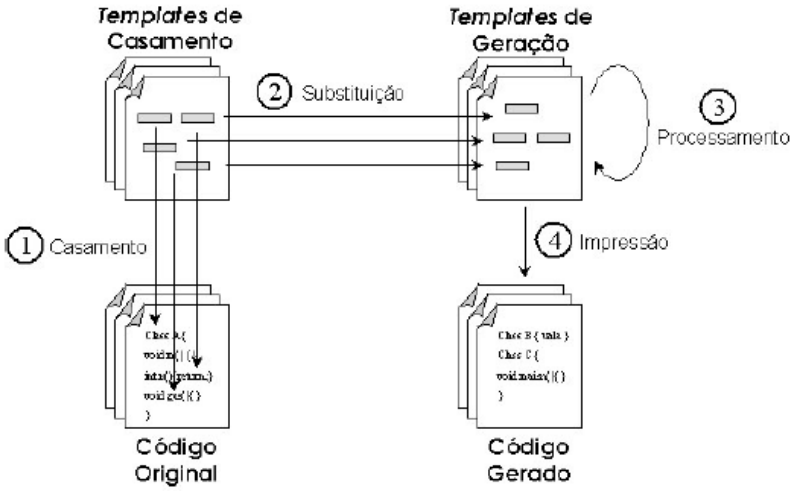

Figura 2.1 Funcionamento de JaTS

Além de JaTS-TL, o sistema JaTS [COS+01, dS06] dispõe de outras duas linguagens domínio específico próprias para a realização de transformações em programas Java:

JaTS-AL JaTS Analisys Language. Linguagem para análise de código.

JaTS-ML JaTS Metaprogramming Language. Linguagem para manipulação de meta-estruturas, definição e aplicação de transformações e especificação de estratégias.

A linguagem JaTS-TL é a mais importante deste sistema, pois é aquela em que os *templates* são codificados. Neste capítulo, introduziremos as construções de JaTS-TL. É importante que se conheça as estruturas desta linguagem, pois os sistemas gerados nesta abordagem, possuem uma linguagem de *templates* que contém os mesmos tipos de construções que JaTS-TL apresenta.

As construções em JaTS-TL são classificadas da seguinte forma:

Construções de casamento Representam elementos do programa Java e podem aparecer tanto nos templates de casamento, quanto nos *templates* de geração.

Construções de geração Servem para manipular diretamente a árvore sintática do programa Java a ser transformado. Estas estruturas só podem aparecer no lado direito pois não representam elementos do programa Java.

2.1 VARIÁVEIS

A construção mais simples de JaTS-TL é a variável. Variáveis servem para armazenar informações sobre os programas fontes. Elas são usadas num primeiro momento para extrair informações do programa fonte e, num segundo momento, para permitir a manipulação da árvore sintática do programa fonte, gerando o programa destino.

Em JaTS-TL, variáveis podem representar quase todas as estruturas de um programa Java, podendo aparecer em todos os pontos de um *template*. Variáveis possuem um tipo que define o tipo de estrutura do programa Java a que podem ser mapeadas. A declaração do tipo só não é obrigatória nos contextos em que o tipo pode ser inferido automaticamente. Como é o caso da variável que representa o nome de uma classe, cujo tipo é sempre um identificador.

Os tipos associados às variáveis JaTs-TL podem ser desde estruturas simples como identificadores até tipos mais complexos como o conjunto de todas as declarações de método de uma classe. Como são responsáveis pelo trânsito de informação entre os dois lados da transformação, variáveis podem estar presentes tanto no *template* fonte quanto no *template* destino.

A notação usada para descrever uma variável JaTS-TL é um identificador Java válido precedido do símbolo `#`. Eventualmente, uma variável pode ser precedida pelo seu tipo:

```
Identifier:#C;
```

ou, quando o tipo puder ser automaticamente detectado:

```
#C
```

Neste exemplo a seguir, tornamos uma classe serializável, através da implementação da interface `java.io.Serializable`. A variável `#C`, casa com o identificador `Conta`.

Template do lado esquerdo:

Listing 2.1 Template de casamento simples

```
public class #C { }
```

Classe de entrada:

Listing 2.2 Classe de entrada

```
public class Conta { }
```

Template de saída:

Listing 2.3 Template de geração simples

```
public class #C implements java.io.Serializable {}
```

Classe resultante:

Listing 2.4 Classe resultante

```
public class Conta implements java.io.Serializable { }
```

2.2 CLÁUSULAS OPCIONAIS

Alguns elementos presentes na classe a ser transformada podem ser irrelevantes ao contexto da transformação. Assim, na descrição de um *template* para a realização de tal transformação, é interessante ter flexibilidade para expressar que um determinado trecho do programa é opcional.

A construção de JaTS-TL que provê esta flexibilidade é a **cláusula opcional**, que serve para especificar que o casamento é opcional para determinados trechos do *template*. Esta construção somente é permitida para aquelas construções da gramática Java que são opcionais: lista de modificadores, declaração da superclasse, lista de interfaces implementadas (ou estendidas, no caso de interfaces), cláusula *throws* de métodos e construtores, uso da palavra reservada *this*. Cláusulas opcionais estão intrinsecamente ligadas à semântica da *linguagem objeto*[1], pois, se não fosse assim, poderíamos obter programas sintaticamente inválidos.

Sintaticamente, uma construção é dita opcional se ela se encontra delimitada pelos símbolos: ‘‘`#[`’’ e ‘‘`]#`’’. No caso específico do `this`, tais delimitadores são dispensados, primeiro, para melhorar a legibilidade do código e segundo porque, no que concerne

[1] *linguagem objeto* é a linguagem manipulada pelo sistema de transformação. Em JaTS é a linguagem Java

a transformação, ele é sempre opcional. Portanto, a expressão:

```
this.#exp;
```

Deve casar tanto com uma expressão:

```
this.metodo();
```

Quanto, com a expressão em que o `this` foi omitido.

```
metodo();
```

Por serem construções de casamento, cláusulas opcionais podem estar presentes tanto no *template* do lado esquerdo quanto no *template* do lado direito da transformação. E indicam que o casamento pode ser realizado tanto na presença quanto na ausência do elemento opcional. No lado esquerdo, cláusulas opcionais indicam que o elemento pode ou não estar presente que o casamento será realizado com sucesso. No lado direito, indicam que, se houver valor mapeado à variável marcada como opcional, a estrutura será produzida na saída. Caso não, nada será produzido.

Aqui, voltamos ao exemplo anterior, porém flexibilizamos o *template*, permitindo que a classe a ser transformada possua superclasse. A variável `#C`, casará com o identificador Conta e a variável `#SC` casará com a superclasse se esta existir.

Template do lado esquerdo:

Listing 2.5 Template de casamento com uso de cláusula opcional

```
public class #C #[extends #SC]# {  }
```

Template de saída:

Listing 2.6 Template de geração com uso de cláusula opcional

```
public class #C #[extends #SC]# implements java.io.Serializable {
```

Classe de entrada sem superclasse definida:

Listing 2.7 Classe de entrada sem superclasse

```
public class Conta {

}
```

Classe resultante:

Listing 2.8 Classe resultante sem superclasses

```
public class Conta implements java.io.Serializable {

}
```

Se a classe de entrada possuir superclasse definida:

Listing 2.9 Classe de entrada com superclasse

```
public abstract class Conta extends ContaAbstrata {

}
```

A mesma transformação irá gerar o seguinte resultado:

Listing 2.10 Classe resultante com superclasse

```
public abstract class Conta extends ContaAbstrata
                    implements java.io.Serializable {

}
```

2.3 DECLARAÇÕES EXECUTÁVEIS

Em JaTS, transformações não são meramente baseadas em casamento e substituição, mas também na manipulação direta da árvore sintática, através de código Java sobre os nós da árvore. Isto permite que JaTS consiga expressar transformações mais complexas do que aquelas permitidas pela simples substituição de valores.

As construções de geração são chamadas de declarações executáveis, estruturas que devem ser executadas a fim de se produzir um programa válido em Java. Elas são

classificadas em (*i*) expressões executáveis, (*ii*) declarações condicionais e (*iii*)declarações iterativas. Nas próximas seções descrevemos cada uma delas.

2.3.1 Expressões Executáveis

Expressões executáveis são estruturas que permitem manipular a árvore sintática do programa que está sendo transformado através da invocações de operações sobre os nós da árvore. Assim, elas permitem a extração ou modificação de informação desses nós.

Expressões executáveis podem ser utilizadas em praticamente todos os contextos em que uma variável pode ser utilizada, exceto pelo fato de que só podem aparecer no *template* de geração. Essas expressões podem ser de dois tipos: Extração de Informação ou Modificação de Informação.

Com o uso de expressões executáveis é possível saber se um determinado atributo é público ou, até mesmo, mudar a visibilidade deste atributo, entre outras operações. Tal recurso é particularmente útil para a criação de novos nós sintáticos a partir de informações obtidas de outros nós.

O exemplo a seguir mostra como um novo nó sintático pode ser formado (declaração de método), utilizando-se informações de outros nós (declaração de atributo `#att` através de expressões executáveis(`#<...>#`).

Listing 2.11 Expressão executável para extração de informação.

```
public class #C {

    public #<#att.getType()># #<#att.addSuffix("get")># () {
        return this.#<#att.getName()>#;
    }
}
```

Se o *template* acima estivesse sendo usado numa transformação, cujo conjunto-resultado fosse: {`#C` → `Conta`,`#att` → `"public int x;"`}, o resultado seria o seguinte:

Listing 2.12 Classe resultante do uso das expressões executáveis

```
    public int getX () {
        return this.x;
    }
}
```

Neste outro exemplo, mudamos a visibilidade do atributo mapeado à variável `#att` para `private`:

Listing 2.13 Expressão executável para modificação de informação.

```
public class #C {

  FieldDeclaration:#<#t = #att :: #t.changeAccessModifierTo("private");>#;
}
```

A exemplo do que acontece com as variáveis JaTS, sempre que não for possível inferir, o tipo da expressão executável deve ser declarado explicitamente.

2.3.2 Declarações Condicionais

Apesar de JaTS definir pré-condições para que as transformações aconteçam, freqüentemente, nos deparamos com situações em que a transformação de apenas um trecho do código é que está condicionada à satisfação de uma condição e não todo o código. Para isso, JaTS-TL define um mecanismo bastante similar a um comando *if-then-else* de Java chamado de declaração condicional. Não apenas o funcionamento, mas também a sintaxe é bastante similar. Exceto pelo demarcador # precedendo a palavra `if`. Assim, uma declaração condicional em JaTS-TL é denotada pelo comando `#if`.

A expressão contida no `#if` é avaliada e, se for satisfeita o código associado ao `#if` será gerado. Caso contrário, se houver uma cláusula `else`, o corpo desta será gerado, se não houver, nenhum resultado é produzido.

Declarações condicionais podem aparecer no corpo da classe ou dentro do corpo de métodos e construtores. O local onde a declaração condicional aparece, define o tipo de construção que pode aparecer dentro de seu corpo, pois, o código gerado por uma declaração condicional deve ser válido sintaticamente no escopo em que foi declarado.

Assim, se a declaração aparece no corpo de uma classe, o código gerado a partir de uma declaração condicional pode ser: declaração de construtor, de método, de atributo e também bloco de inicialização. Se a declaração, no entanto, aparece dentro de um método ou construtor, o código gerado só pode conter declarações (*statements*) que possam aparecer no corpo de um método, como: bloco, declaração de variável local, *if-then-else*, *switch*,

for, etc.

Por se tratar de uma construção de transformação, uma declaração condicional só pode aparecer no *template* destino da transformação.

A sintaxe de uma declaração condicional de JaTS é a seguinte:

```
 #if ( <condicao> ) {
    // código a ser gerado, caso <condicao> seja avaliada para true
} else {
   // código a ser gerado, caso <condicao> seja avaliada para false
}
```

A expressão denotada por `condicao` deve ser booleana. Condição tanto pode ser uma variável, neste caso, do tipo `Literal`, quanto pode ser uma expressão executável cujo tipo de retorno seja booleano.

Neste exemplo, casamos uma classe que contém um único atributo. Se este atributo for publico, tornamo-lo privado e adicionamos o método de acesso correspondente. Se não for, apenas o escrevemos no programa de saída.

Lado esquerdo:

Listing 2.14 Template de casamento

```
public class #C {
    FieldDeclaration:#att;
}
```

Lado direito:

Listing 2.15 Template de geração usando declaração condicional

```
public class #C {

  #if(#<#att.isPublic()>#) {

    FieldDeclaration:#<#tmp = #att ::
                        #tmp.changeAccessModifierTo("private");>#;

    public #< #att.getType() >#
           #< #att.getName().addPrefix("get") >#() {
      return this.#< #att.getName() >#;
```

```
    }
  }
  else {
    FieldDeclaration:#att;
  }
}
```

Se esta transformação fosse aplicada ao programa:

Listing 2.16 Classe de entrada contendo atributo

```
public abstract class ContaAbstrata {
   public String numero;
}
```

O resultado seria:

Listing 2.17 Classe resultante do uso de declarações condicionais

```
public abstract class ContaAbstrata {
   private String numero;

   public String getNumero() {
      return this.numero;
   }
}
```

2.3.3 Declarações Iterativas

JaTS possui variáveis que podem ser mapeadas não somente a um elemento, mas a um conjunto deles. Por exemplo, o conjunto de todas as declarações de atributos de uma classe. Este tipo de construção é particularmente útil, quando queremos escrever uma transformação que adiciona um novo método, mas mantém os atributos de uma determinada classe.

Certas transformações, porém, requerem que algum processamento seja feito nesse tipo de conjunto. Em todos os elementos ou em apenas parte deles. Para isto, JaTS define

uma estrutura de iteração sobre elementos de um conjunto, denominada **Declaração Iterativa**.

Assim como acontece nas declarações condicionais, declarações iterativas podem estar presentes dentro do corpo de classes e interfaces e também dentro do corpo de métodos e construtores e as mesmas restrições a respeito do código gerado se aplicam. Além disto, declarações iterativas e condicionais são aninháveis entre si. Ou seja, dentro de declarações iterativas podemos ter declarações condicionais e vice-versa.

Sintaxe de uma declaração iterativa:

```
forall <var> in <conjunto> {

}
```

A seguir, um exemplo de transformação usando declaração iterativa. Todos os atributos da classe são percorridos e, para cada um deles, é gerado o método `get` correspondente.

Lado esquerdo:

Listing 2.18 Template de casamento com uso de variáveis de conjunto

```
public class #C {
    FieldDeclarationSet:#fds;
    MethodDeclarationSet:#mds;

}
```

Lado direito:

Listing 2.19 Template de geração usando declaração iterativa

```
public class #C {
    FieldDeclarationSet:#fds;
    #forall #v in #fds {
        public #< #v.getType() >#
                #< #v.getName().addPrefix("get") >#() {
            return this.#< #v.getName() >#;
        }
    }
    MethodDeclarationSet:#mds;
}
```

Se esta transformação fosse aplicada ao programa:

Listing 2.20 Classe de entrada com múltiplos atributos

```
public abstract class ContaAbstrata {
   private String numero;
   private double saldo;

   public abstract void debitar(double valor);

}
```

O resultado seria:

Listing 2.21 Classe resultante do uso de declarações iterativas

```
public abstract class ContaAbstrata {
   private String numero;
   private double saldo;

   public abstract void debitar(double valor);

   public String getNumero() {
      return this.numero;
   }

   public String getSaldo(){
       return this.saldo;
   }
}
```

CAPÍTULO 3

SEMÂNTICA DE JATS

Neste capítulo, introduzimos a semântica de JaTS. A semântica é apresentada informalmente. Além disto, como a sintaxe de JaTS-TL é um superconjunto da sintaxe de JaTS, decidimos omitir as construções de Java, já que tal sintaxe encontra-se largamente difundida [GJSB96]. A semântica de Java também não será descrita, uma vez que também esta já foi estudada e documentada [WB99]. Este trabalho é, em parte, uma revisão do que já havia sido feito por Castor [Cas01b].

O entendimento da semântica de JaTS-TL é essencial, pois os sistemas gerados nesta abordagem são baseados em linguagens de templates que contém as mesmas contruções.

A seguinte estrutura será usada para apresentar as construções de JaTS:

- Sintaxe
- Semântica
  - Casamento
  - Substituição
  - Execução

3.1 IDENTIFICADORES, NOMES, LISTA DE NOMES, VARIÁVEIS E EXPRESSÕES EXECUTÁVEIS

3.1.1 Sintaxe

```
Identifier ::= Word | VariableId | ExecutableExpression

Word ::= alpha(alpha|digit)*

VariableId ::= "#"Word

Name ::= Identifier (‘‘.’’ Identifier)*

NameList ::= Name (‘‘,’’ Name)*
```

3.1.2 Semântica

3.1.2.1 Casamento

- O casamento entre dois identificadores pode resultar ou apenas no sucesso, no caso de os dois serem idênticos, ou no mapeamento da variável que representa um identificador ao identificador correspondente ou pode resultar em erro, caso os identificadores não sejam idênticos ou a variável que representa o identificador já estiver casada com outro identificador de valor diferente. O casamento entre "Conta" e "Conta" resulta em sucesso; O casamento entre "`#C`" e "Conta" resulta no mapeamento: "`#C` → `Conta`"; O casamento entre "Conta" e "ContaAbstrata" resulta em erro e o casamento entre "`#C`" e "Conta", no contexto de um conjunto-resultado que contém os seguintes valores {`#C` → `''Endereco''`} resulta em erro, pois "Conta" é diferente de "Endereco".

- Expressões executáveis não podem ser casadas.

- O casamento entre dois nomes consiste em casar um a um os identificadores que os compõem. Na ordem em que estes aparecem. Se o primeiro nome, no entanto, for uma variável, então, o casamento resultará no mapeamento entre esta variável e o nome correspondente. O casamento entre "qib.contas" e "qib.contas" resulta em sucesso. Já o casamento entre "`#pckg`" e "`qib.contas`" resulta no mapeamento: "`#pckg` → `qib.contas`". E o casamento entre "`qib.#name`" e "`qib.contas`" resulta no mapeamento: "`#name` → `contas`".

- O casamento entre duas variáveis não é permitido. É importante que se perceba que isto nunca poderia ocorrer, visto que o casamento é uma operação que envolve um *template* e um programa Java. Como o programa Java não contém variáveis, este casamento nunca irá acontecer.

3.1.2.2 Substituição

- A substituição de qualquer estrutura que não seja uma variável, resulta na própria estrutura.

- A substituição de uma variável resulta sempre na estrutura a qual esta se encontra mapeada. Tal estrutura tem o mesmo tipo sintático da variável. Assim, se a variável

representa um identificador, então, a estrutura que a esta se encontra mapeada é um identificador.

- A substituição de nomes consiste ou na substituição da variável correspondente ao nome a esta mapeado. Ou então, na substituição de cada um dos identificadores que o compõem.
- Listas de nomes têm substituição similar a de nomes, porém estas são compostas por nomes e não por identificadores.

3.1.2.3 Execução

- A execução de um identificador resulta no próprio identificador.
- A execução de uma variável resulta na própria variável.
- A execução de uma expressão executável consiste em na interpretação das expressões contidas entre os delimitadores da expressão executável(“`#<`” e “`>#`”) , similar ao que aconteceria se fossem executadas por um interpretador Java. Existem dois tipos de expressões executáveis: a de extração de informação consiste de uma única expressão e seu resultado é o resultado da interpretação desta expressão. A segunda, de modificação de informação, consiste de uma variável, seguida de uma seqüência de expressões e o resultado é a interpretação destas várias expressões , que são associadas à variável que aparece no começo desta expressão.
- A execução de um nome resulta na execução de cada um identificadores que o compõem ou na execução da expressão executável associada.
- A execução de uma lista de nomes resulta na execução de cada um nomes que a compõem ou na execução da expressão executável associada.

3.2 TIPOS

3.2.1 Sintaxe

```
Type ::= PrimitiveType | Name | ‘‘Type’’ ‘‘:’’ VariavelId  |
        ‘‘Type’’ ‘‘:’’ ExecutableExpression

PrimitiveType ::= "boolean"|"byte"|"char"|"short"|"int"|
"long"|"float"|"double"
```

3.2.2 Semântica

3.2.2.1 Casamento

- Dois tipos irão casar se forem tipos primitivos, ou forem representados por classes (`Name`), ou quando um for variável e o outro for um tipo primitivo ou uma classe.
- No caso de os tipos serem primitivos, simplesmente checa-se se representam o mesmo tipo. Ou seja, se os dois são "`int`" ou "`char`", etc.
- No caso de serem representados por classes, os tipos são denotados por `Name` e, vale aqui o mesmo que para nomes.
- Se um deles é variável, então o resultado será o mapeamento desta variável para o tipo correspondente.

3.2.2.2 Substituição

- A substituição de tipos ocorre de maneira similar a de identificadores e nomes, já citada anteriormente.

3.2.2.3 Execução

- A execução de tipos ocorre de maneira similar a de identificadores e nomes, já citada anteriormente.

3.3 EXPRESSÕES

3.3.1 Sintaxe

A sintaxe de expressões em JaTS-TL é um superconjunto das expressões da linguagem Java. A descrição das expressões de Java serão omitidas e, em seu lugar, usaremos o não-terminal `JavaExpression` a fim de para facilitar a compreensão. Somente as produções modificadas serão descritas. Se o leitor considerar necessário, pode consultar a especificação da linguagem Java [GJSB96].

```
JaTSExpression ::= JavaExpression | VariableId |
ExecutableExpression

Literal::= StringLiteral | IntegerLiteral | FloatingPointLiteral|
BooleanLiteral | NullLiteral| Variable | ExecutableExpression
```

```
ExpressionList::= Expression ( "," Expression)*
```

3.3.2 Semântica

3.3.2.1 Casamento

- O casamento de dois literais pode resultar simplesmente em sucesso (no caso de serem idênticos) ou pode resultar no mapeamento entre a variável e o literal correspondente.
- O casamento de literais resulta em sucesso se estes forem idênticos. Não há , até o presente momento, variáveis que representem operadores.
- O casamento de lista de expressões consiste em casar uma a uma e na ordem em que aparecem as expressões que a compõem.
- O casamento das demais expressões consiste no casamento de cada parte que a compõe.

3.3.2.2 Substituição

- A substituição de literais é similar à substituição de identificadores.
- A substituição de operadores resulta no próprio operador.
- A substituição de uma lista de expressões resulta na substituição da variável que representa toda a lista ou na substituição de cada uma das expressões que a compõem. Por exemplo, a substituição de `(a+3,#exp, #expr2)`, dado que existem os mapeamentos {`#exp`→`(pow(4,2))`, `#expr2` → `6.4`}, resulta em: `(a+3, (pow(4,2)),6.4)`.
- A substituição das demais expressões é feita substituindo-se cada um dos elementos que a compõem. No caso de uma expressão binária, corresponde no casamento da expressão no lado esquerdo, no casamento do operador e, em seguida, no casamento da expressão do lado direito.

3.3.2.3 Execução

- A execução de um literal é similar a de um identificador. Se este for um literal de Java, terá como retorno, o próprio literal. Se a este literal estiver associado uma expressão executável, a execução é a mesma descrita na seção sobre identificadores, nomes e expressões executáveis(Seção 3.1)

- Expressões contidas dentro de expressões executáveis são executadas e seu valor é retornado para o nó de origem. Expressões que aparecem ao longo do código que não estejam embutidas em uma expressão executável é apenas retornada. As expressões são executadas por meio de reflexão.

- A execução de um acesso a atributo ([<expressao>.]<nome>) consiste na execução da expressão referente ao objeto referenciado, caso ela exista, seguida da avaliação do nome correspondente ao atributo.

- A execução de uma chamada a método (<`expressao`>.]<`nome`>(<`argumentos`>)) consiste na avaliação da expressão referente ao objeto referenciado, seguida da avaliação dos argumentos e, por fim, a execução do método denotado por <nome>, com os parâmetros denotados por <argumentos>.

- A execução de uma `InstanceOfExpression` resulta na execução do objeto sendo consultado, seguido da execução do tipo associado e, por fim, a avaliacao se o tipo do objeto consultado é compatível com o tipo expresso na consulta.

- A execução de uma expressão unária consiste na execução do operando expressão, seguido da aplicação do operador ao operando. Uma expressão "`!(#expr1)`", no contexto do seguinte conjunto-resultado: "`#expr1 → "true"`", consiste da avaliação de `#expr1 (true)`, seguida da aplicação do operador "`!`" ao operando: `!(true) = false`. A execução de expressões que dependam de contexto de execução (incremento pré e pós fixado) não é suportada por JaTS.

- A execução de uma expressão binária consiste na execução do lado esquerdo da expressão, seguido da execução do lado direito, por fim, consiste no resultado da aplicação do operador aos operandos. Uma expressão "`#expr1 + #expr2`", no contexto do seguinte conjunto-resultado: "`#expr1 → 4, #expr2 → 7+9` ", consiste da avaliação de `#expr1` (4), seguida da avalição de avaliação de `#expr2` (16) e depois da aplicação do operador "`+`" aos dois operandos: $(4) + (16) = 20$. A execução de expressões que dependam de contexto de execução (atribuição) não é suportada por JaTS.

- A execução de uma expressão ternária (<expr1>?<expr2>:<expr3>) consiste na execução da expressão referente a condição (<expr1>). Caso a expressão que denote a condição seja avaliada como verdadeira, entao, o resultado será a execução da expressão <expr2>. Caso seja avaliada para false, o resultado será a execução denotada por <expr3>.

- A execução de uma lista de expressões consiste em avaliar cada uma das expressões, na ordem em que aparecem.

3.4 DECLARAÇÕES

3.4.1 Sintaxe

A sintaxe de declarações em JaTS é muito similar a de Java, com o acréscimo de variáveis e de produções que representam construções de JaTS. Nesta seção iremos citas apenas algumas aquelas necessárias ao entendimento do funcionamento de JaTS.

```
ClassBodyDeclaration ::= FieldDeclaration
                        |FieldVarDeclaration
                        |FieldsVarDeclaration
                        |MethodDeclaration
                        |MethodVarDeclaration
                        |MethodsVarDeclaration
                        |ConstructorDeclaration
                        |ConstructorVarDeclaration
                        |ConstructorsVarDeclaration
                        |IterativeDeclaration
                        |ConditionalDeclaration

FieldVarDeclaration :: = ‘‘FieldDeclaration’’ ‘‘:’’ VariableId;
FieldsVarDeclaration :: = ‘‘FieldDeclarationSet’’ ‘‘:’’ VariableId;

IterativeDeclaration ::= ‘‘forall’’ VariableId ‘‘in’’VariableId
‘‘{’’ClassBodyDeclaration* ‘‘}".


ConditionalDeclaration ::= ‘‘#if’’( Expression )
                        ‘‘{’’ ClassBodyDeclaration* ‘‘}’’
```

```
["else’’‘‘{’’ClassBodyDeclaration* ‘‘}’’]
```

3.4.2 Semântica

3.4.2.1 Casamento

- Para todas as declarações herdadas da linguagem Java, o casamento funciona da mesma forma: O casamento entre duas declarações de mesmo tipo (por exemplo, duas declarações de método), resulta no casamento de todas as estruturas que as compõem. No caso do casamento entre duas declarações de método, significa casar os modificadores, os tipos de retorno, o identificador do método, a lista de argumentos, a lista de exceções e o corpo do método.

- Duas declarações só casam se possuem o mesmo tipo. Qualquer outra combinação irá resultar em falha do casamento.

- Declarações do tipo `IterativeDeclaration` não podem ser casadas. Por serem declarações executáveis, elas não podem estar presentes no programa JaTS-TL a ser casado.

- Declarações do tipo `ConditionalDeclaration` não podem ser casadas. Por serem declarações executáveis, elas não podem estar presentes no programa JaTS-TL a ser casado.

O casamento das declarações: `FieldVarDeclaration`, `FieldsVarDeclaration` constitui um dos casamentos mais interessantes de JaTS.

De forma a simplificar o entendimento, usaremos o casamento de atributos, como exemplo. As regras para este tipo de casamento, também se aplicam aos demais tipos de declaração da linguagem Java, tais quais construtores, métodos, declarações de importação.

A ordem do casamento é a seguinte:

1. Primeiro, realiza-se o casamento dos itens mais específicos. Ou seja, as declarações completas de atributo (No formato: `<modificador><tipo><variaveis>`) existentes no *template* com as declarações no programa Java. As declarações no programa Java que já foram casadas, são removidas do total de declarações possíveis de serem casadas.

2. Casam-se as variáveis do tipo `FieldDeclaration` na ordem em que aparecem. Ou seja, a primeira variável desse tipo, casa com a primeira declaração de atrib-

uto ainda não casada presente no programa Java. Não podem sobrar variáveis `FieldDeclaration` sem serem casadas.

3. Realiza-se, então, o casamento das variáveis do tipo FieldDeclarationSet com as declarações de atributo restantes no programa Java. Se houver mais de uma, somente a primeira é casada. As demais, são mapeadas a {}, onde {}, indica um conjunto vazio.

3.4.2.2 Substituição

- A substituição das declarações herdadas de Java consiste em substituir, seqüencialmente, todos as estruturas que as compõem.
- A substituição de declarações condicionais consiste na substituição de todas as estruturas que a compõem.
- Numa declaração ‘‘`forall`’’ `#fd` ‘‘`in`’’ `#fds` ‘‘`{`’’ `IterativeBodyDeclaration`‘‘`}`’’ , apenas a variável referente `#fds` é substituída.

3.4.2.3 Execução

- A execução de todas as declarações herdadas de Java consistem na própria declaração.
- A execução de uma declaração condicional é feita primeiramente avaliando-se a condição. Caso seja verdadeira, o resultado da execução é o resultado da execução do bloco associado ao *#if*. Caso contrário, o resultado da execução é igual ao resultado da execução do bloco *else* se ele existir, caso não exista, o resultado é um conjunto vazio.

Para a execução da declaração:

‘‘`forall`’’ `<var>` ‘‘`in`’’ `<conjunto>` ‘‘`\{`’’ `IterativeBodyDeclaration` ‘‘`\}`’’

vamos supor que `<conjunto>` é uma lista `L`. E que temos três operações sobre `L`: `head()`, `isEmpty()` e `tail()`, que retornam, respectivamente, o primeiro elemento da lista `L`, um booleano indicando se a lista `L` está vazia ou não e a lista sem o primeiro elemento.

Os passos, então, são os seguintes:

1. Se a lista estiver vazia, isto é, se `isEmpty(L)` retornar verdadeiro, retorna sem executar nada.

2. Adiciona-se ao conjunto-resultado, o mapeamento `(var, head(L))`.
3. Realiza-se a execução do corpo do `forall`.
4. Remove-se o mapeamento `(var, head(L))` do conjunto.
5. Atualiza-se `L`, de forma a remover o primeiro elemento: `L = tail(L)`.
6. Repete-se até os passos anteriores.

3.5 COMANDOS

3.5.1 Sintaxe

Nesta seção listamos apenas as produções que foram alteradas para a reeber novas derivações relativas aos comandos introduzidos por JaTS-TL.

```
Statement := LabeledStatement | IfStatement | WhileStatement |
             ForStatement | Block | EmptyStatement|
             ExpressionStatement | SwitchStatement | DoStatement |
             BreakStatement | ContinueStatement | ReturnStatement |
             SynchronizedStatement | ThrowStatement | TryStatement |
             IterativeStatement | ConditionalStatement |
             StatementList | LetStatement

IterativeStatement :=  ‘‘forall’’ VariableId ‘‘in’’
                    ( VariableId | ExecutableExpression)
                    Block

ConditionalStatement := ‘‘#if’’ ‘‘(’’( VariableId
                                     |ExecutableExpression)
                         [ ‘‘else’’ Block ]

StatementList := ‘‘StatementList’’ ‘‘:’’ VariableId ‘‘;’’

LetStatement := ‘‘let’’
                ( VariableId VariableInitializer ‘‘;’’)+ ‘‘in’’
                Statement
```

3.5.2 Semântica

3.5.2.1 Casamento

- O casamento dos comandos herdados da linguagem Java são feitos de forma similar ao casamento de declarações. Só comandos de mesmo tipo podem casar. Assim, um comando **if-then-else** só pode casar com outro comando de mesmo tipo. Para que o casamento ocorra com sucesso, todas as estruturas que compõem o comando devem ser casadas, na ordem em que aparecem.
- O casamento entre comandos diferentes (ex: `If` e `While`) resulta em falha.
- O casamento de um bloco consiste no casamento de todos os comandos que o compõem, na ordem em que estes aparecem.
- Comandos do tipo `IterativeStatement` não podem ser casados.
- Comandos do tipo `ConditionalStatement` não podem ser casados.
- Comandos do tipo `LetStatement` não podem ser casados.
- O casamento de comandos do tipo `StatementList` está completamente descrito em [dS06].

3.5.2.2 Substituição

- A substituição dos comandos herdados de Java consiste da subsistuição das estruturas que os compõem.
- A substituição de `IterativeStatement` consiste apenas na substituição de valores da expressão que denota o conjunto a ser iterado.
- A substituição de `ConditionalStatement` consiste na substituição do expressão que denota a condição, bem como na substituição de cada estrutura que o compõe.

3.5.2.3 Execução

- A execução dos comandos herdados de Java são similares à execução de declarações. Resultam no próprio comando sendo retornado.
- A execução de um bloco consiste na execução dos comandos presentes dentro do bloco na ordem em que aparecem.

- A execução de um `IterativeStatement` é similar à execução de uma `IterativeDeclaration`, exceto pelo fato de que o resultado da execução é um conjunto de comandos, ao invés de um conjunto de declarações.

- Analogamente, a execução de um `ConditionalStatement` é similar à execução de uma `ConditionalDeclaration`, exceto pelo fato de que o resultado da execução é um conjunto de comandos, e não um conjunto de declarações.

- A execução de uma declaração *let* como a seguinte: `let #varName = #<#A.getName()>#;` consiste na execução da expressão associada a variável que está sendo introduzida (`#<#A.getName()>#`) e em seguida, na adição do par (variável, resultado da avaliacao da expressão) no conjunto-resultado. Supondo que exista um conjunto-resultado com o seguinte mapeamento "`#A` → `"private double saldo;"`". A execução da declaração let : `let #varName = #<#A.getName()>#;`, traria como resultado um novo conjunto-resultado: `#A` → `"private double saldo;", #varName` → `saldo`". Depois de modificar o conjunto-resultado, o comando associado é executado.

CAPÍTULO 4

GERAÇÃO DE SISTEMAS DE TRANSFORMAÇÃO

I would rather write programs that help me write programs than write programs

—DICK SITES

À medida que os sistemas se tornam mais complexos, cresce a necessidade de desenvolver componentes em mais de uma linguagem. O desenvolvimento de um sistema Web, por exemplo, envolve pelo menos três linguagens: a linguagem de desenvolvimento do núcleo do sistema (por exemplo, Java [GJSB96]), a linguagem de apresentação (por exemplo, JSP [Hal00]+ HTML [W3C05]) e a linguagem usada para configuração (por exemplo, XML [HM01]).

Portanto, cresce também a necessidade de ferramentas que gerem e mantenham códigos escritos em várias linguagens. Sem isso, a produtividade dos desenvolvedores pode diminuir, tornando alto o custo de se fabricar e manter softwares complexos.

Para diminuir este custo, o ideal é contar com ferramentas que auxiliem no processo de construção (através de geração automática) e de manutenção (através de transformações, refatorações, etc.). Entretanto, como tais sistemas são codificados em mais de uma linguagem, o ideal é que tais ferramentas possam transformar todas as partes do sistema, ou seja, que suportem mais de uma linguagem. Para isto, temos duas alternativas: sistemas de transformação independentes de linguagem ou vários sistemas de transformação para cada linguagem envolvida nesses sistemas.

Os sistemas independentes de linguagem têm a vantagem de transformar programas escritos em várias linguagens. Porém, esta generalidade traz o inconveniente de não poder expressar transformações complexas e também de requerer que a gramática da linguagem seja entrada para a transformação. Já os sistemas específicos têm a capacidade de expressar de forma mais simples, transformações complexas e transformações que requerem conhecimento da semântica da *linguagem objeto*.

Assim, para evoluir sistemas complexos, o cenário ideal seria dispor de um conjunto de sistemas específicos para cada linguagem de que o sistema faz uso. Porém, para facilitar a assimilação por parte do programador, o ideal seria que estes sistemas tivessem as mesmas características.

Em nosso trabalho, definimos uma abordagem para geração de sistemas de transformação usando técnicas de programação gerativa [CE00]. Assim, podemos unir as vantagens de se ter um sistema de transformação específico para cada linguagem com a vantagem de todos os sistemas terem um único modelo de funcionamento e serem baseados em linguagens de templates com construções similares.

O JaTS [CB01, COS+01] é usado como modelo para a linha de produtos de sistema de transformação. Além disto, o JaTS é usado como ferramenta de geração de código para tais sistemas e partes da sua implementação são reusadas por estes sistemas.

Como o JaTS é usado como sistema modelo, nosso primeiro passo foi avaliar a arquitetura de JaTS e os aspectos de sua implementação a fim de identificar pontos que pudessem ser reusados e, também, identificar a estrutura das classes que seriam geradas automaticamente.

Portanto, começamos este capítulo com uma breve explicação sobre a arquitetura de JaTS, depois mostramos as refatorações realizadas no sistema JaTS atual a fim de torná-lo mais flexível e reusável e, por fim, nos dedicamos à geração propriamente dita.

4.1 O SISTEMA JATS

Como visto no Capítulo 2, JaTS é um sistema composto por três linguagens: JaTS-TL, JaTS-ML e JaTS-AL. Este trabalho, entretanto, só aborda a primeira delas. A última, JaTS-ML, é uma linguagem de metaprogramação que pode ser utilizada para manipular transformações de qualquer sistema gerado por este trabalho.

Apenas relembrando o que já foi dito no Capítulo 2 em que explicamos a semântica da linguagem, os passos de uma transformação em JaTS são os seguintes:

1. Geração das árvores sintáticas dos templates e programas Java passados como parâmetro (*parsing*);

2. Casamento do *template* do lado esquerdo com o programa fonte, gerando um conjunto de mapeamentos denominado conjunto-resultado (casamento);

3. Substituição das variáveis presentes no lado direito pelos valores que a estas se encontram mapeados no conjunto-resultado (substituição);

4. Processamento das estruturas executáveis presentes no lado direito, a fim de obter uma árvore sintática válida em Java (processamento);

5. Impressão da árvore sintática resultante (*prettyprinting*).

Os passos inicial e final dizem respeito a operações corriqueiras de I/O, dentro de um sistema de transformação, como por exemplo, um compilador. Enquanto os passos intermediários são responsáveis pela transformação propriamente dita. Por isso, na arquitetura de JaTS, estão presentes dois módulos: O primeiro, responsável pelo *parsing* e *pretty-printing* e o segundo, pelas operações de casamento, substituição e processamento. A Figura 4.1 ilustra como estes módulos estão dispostos na arquitetura de JaTS.

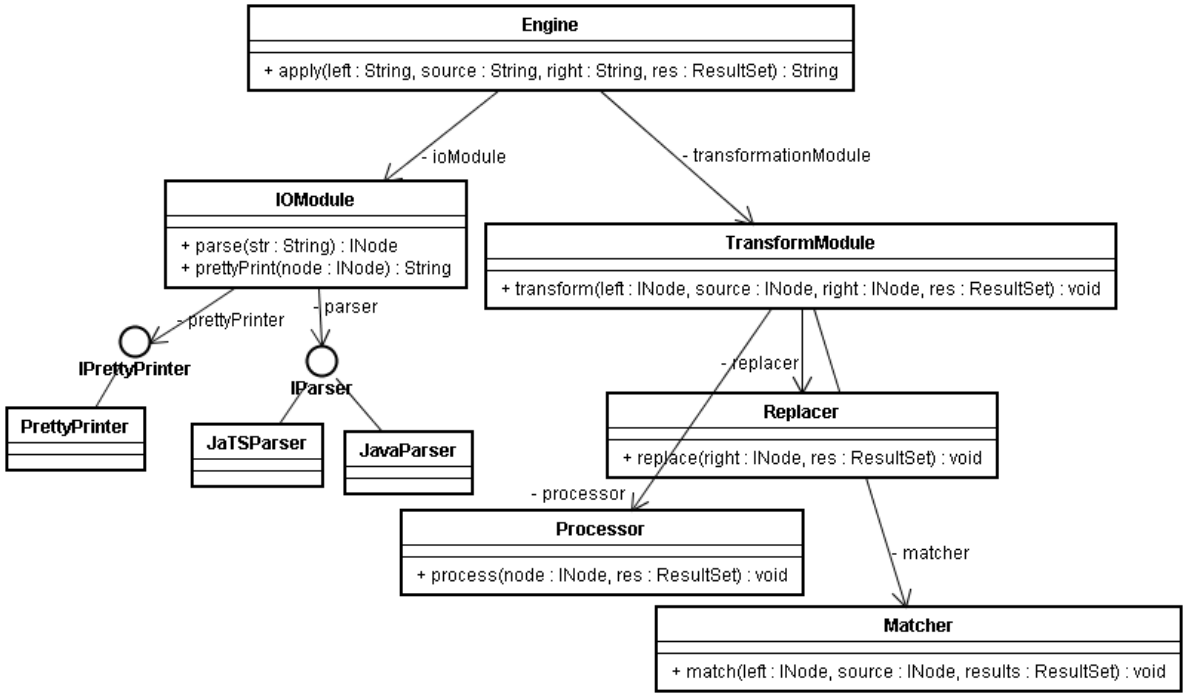

Figura 4.1 Arquitetura de JaTS

O módulo de I/O está diretamente ligado à *linguagem objeto* e, por esta razão, pouca coisa, ou nenhuma, pode ser reaproveitada de um sistema para outro, a não ser as interfaces para o *parser* e para o *prettyprinter*. O módulo de transformação, porém, é fortemente baseado na interface `INode`, que deve ser implementada por qualquer classe que represente um nó da árvore sintática. Assim sendo, as suas funcionalidades podem ser aplicadas a qualquer classe que implemente `INode`, não importando a que linguagem o nó pertence. Esta interface é a responsável por ligar os dois módulos, levando informação entre um e outro.

Na forma como foi implementado, todos as classes que representam nós da árvore sintática de JaTS, implementam a interface `INode`, pois é nos nós sintáticos que reside a responsabilidade pela transformação propriamente dita. Os módulos presentes em `TranformModule` apenas delegam aos nós sintáticos a responsabilidade pelas operações

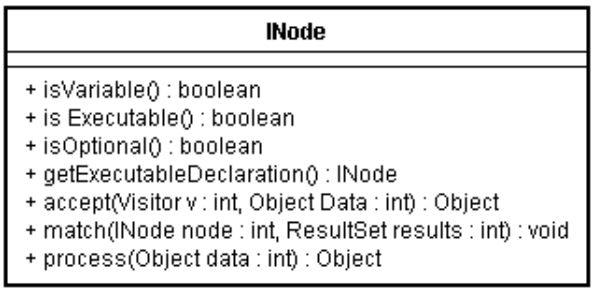

Figura 4.2 Interface dos nós sintáticos

de casamento, substituição e processamento. Sendo tão importante neste sistema, é fundamental que se observe os métodos presentes nesta interface, conforme a Figura 4.2.

As operações de casamento, substituição e processamento necessitam de caminhamento dentro da árvore sintática. Por exemplo, uma vez que se realiza o casamento entre nós que representam uma classe em Java (`ClassDeclaration`), é preciso que cada um de seus filhos, por exemplo, as declarações de atributos presentes (`FieldDeclarationSet`), sejam casados também.

Foram escolhidas duas estratégias para a implementação destas operações: uma para navegação no sentido *top-down* e outra no sentido *bottom-up*. Isto porque algumas operações demandam uma consulta prévia ao nó atual para saber se vão ou não operar sobre os filhos deste nó. Então, navegar no sentido *top-down* evita processamento desnecessário. Já outras operações devem ser executadas nos filhos de qualquer forma e é interessante que sejam executadas antes nos filhos para que os pais possam usar o resultado. Tais operações foram implementadas usando, essencialmente, os padrões de projeto: *Interpreter* e *Visitor* [GHJV94].

As operações que usam *Interpreter* são as de casamento (*match*) e processamento(*process*). No caso da operação de casamento, a escolha do padrão foi feita em função de esta ser bastante dependente da estrutura interna do nó sintático, permitindo uma maior coesão. No caso da operação de processamento, o padrão é o mais natural, visto que esta operação é usada na interpretação de expressões executáveis.

O padrão *Visitor* foi usado na operação de substituição e, por este motivo, esta operação não aparece na interface `INode`. Ao invés disso, aparece o método *accept*. Este método é o responsável por navegar na árvore, no sentido *bottom-up* e aplicar uma operação. Uma das operações possíveis é a de substituição. Mas outras duas operações também usam este padrão: clonagem de nós e a impressão. Uma vantagem no uso deste padrão é o fato de que novas operações podem ser realizadas sobre os nós sem que seja preciso alterar suas estruturas.

Os demais métodos presentes na interface `INode` são auxiliares na transformação e servem para dar informações sobre o nó em questão, por exemplo se ele é executável ou não (`isExecutable()`). No caso de nós executáveis, é possível obter a declaração executável associada (`getExecutableDeclaration()`).

4.1.1 Reestruturação do código de JaTS

Antes de começarmos a usar o JaTS como modelo para os demais sistemas, decidimos avaliar a implementação, com o intuito de identificar pontos de melhoria. Tanto para melhorar a implementação do JaTS atual, quanto para simplificar os sistemas que derivam dele.

Observando outros aspectos da implementação, identificamos a existência de quatro tipos de nós sintáticos em JaTS:

- Classes que representam um único nó. Por exemplo, `FieldDeclaration`;
- Classes que representam uma lista de nós (a ordem dos elementos importa). Por exemplo, `ParameterList`;
- Classes que representam um conjunto de nós (a ordem não é importante). Por exemplo, `FieldDeclarationSet`;
- Classes que representam um conjunto de declarações dentro do corpo de uma declaração de classe. E que realizam o casamento independente da disposição das declarações dentro de uma classe. Por exemplo, `BodyFieldDeclarations`.

Entre esses tipos, aqueles que representavam agregações apresentavam um alto grau de replicação de código. Assim, classes como a que representavam o conjunto de atributos ou o conjunto de métodos têm estrutura bastante similar, diferindo apenas pelo tipo de dados que manipulavam. Estas classes, por não poderem usar herança, afinal, conceitualmente, um conjunto de atributos não pode ser filha de um conjunto de métodos e vice-versa, faziam uso do padrão *Decorator*. Mesmo delegando parte do trabalho a uma classe comum, boa parte do código entre elas, era muito semelhante. Até a versão *1.4* da linguagem Java, pouca coisa poderia ser feita para resolver este problema. Porém, dado que a versão *1.5* [GJSB05] introduziu o conceito de *tipos genéricos*[1], foi possível remover uma parte substancial da replicação.

[1] Uma classe é dita genérica se ela é parametrizada por uma ou mais variáveis de tipo. Uma declaração de classe genérica define um conjunto de classes, parametrizadas por tipos, que compartilham a mesma definição de classe em tempo de execução.

As classes que representavam conjuntos e aquelas que representavam listas, já usavam delegação a classes auxiliares. Portanto, isso facilitou parte deste processo.

No caso das listas, foi preciso apenas fazer com que a classe `NodeList` se tornasse genérica e que seu parâmetro de tipo, fosse descendente de `INode`. Assim, temos:

Listing 4.1 Classe NodeList

```
public class NodeList<T extends INode> {...}
```

As classes que representavam listas não foram eliminadas, pois definiam métodos que lidavam de forma muito específica com o tipo que manipulavam. Assim, estas passaram a herdar da classe `NodeList` parametrizada pelo tipo de interesse com a adição do código específico ao tipo de interesse. Desta forma, o código replicado foi eliminado, porém, as especificidades puderam continuar.

Listing 4.2 Classe ParameterList

```
public class ParameterList extends NodeList<Parameter> {

   FieldDeclarationSet toFieldDeclarationSet() {...}

}
```

Para declarações de conjuntos, usamos uma interface que já estava presente em JaTS, para definir o parâmetro da classe genérica: a interface `Declaration`. Diferente do que acontecia nas listas, os conjuntos puderam ser completamente eliminados, ficando portanto, apenas uma classe para representá-los. Assim, para representar um conjunto de atributos ou um conjunto de métodos, basta instanciar a classe convenientemente.

```
public class GenericDeclarationSet<T extends Declaration> { }

GenericDeclarationSet<FieldDeclarationSet> fieldDeclarationSet =
            new GenericDeclarationSet<FieldDeclarationSet>();

GenericDeclarationSet<MethodDeclarationSet> methodDeclarationSet =
            new GenericDeclarationSet<MethodDeclarationSet>();
```

As classes que representam o conjunto de declarações de um determinado tipo em uma classe, também foram reduzidas a uma só. Desde que estas implementem a interface `Declaration`, a qual define um único método, `getSignature`. Este método é usado para verificar se uma determinada declaração já existe em um conjunto:

Listing 4.3 Interface Declaration

```
public interface Declaration {
   Signature getSignature();
}
```

A interface `Signature`, por sua vez, é uma interface vazia. Ela deve ser implementada de acordo com a declaração a qual está associada. Por exemplo, para uma declaração de atributo, pode ser a própria classe `Identifier`. Porém, para uma declaração de método, usamos a classe `MethodSignature`, que armazena o nome e os parâmetros de um método.

Outras classes utilitárias de JaTS também foram generalizadas, mas o maior ganho foi observado nas classes de agregação, pois além da redução de classes, esta refatoração permitiu simplificar os *Visitors* existentes, uma vez que os métodos relativos às classes que representavam agregações foram substituídos pelo método *visit* referente a cada tipo genérico. No total, a redução foi de **12** classes e **39** métodos, o que corresponde a, aproximadamente, **6k** linhas de código (LOC).

Para ilustrar o ganho obtido com esta refatoração, podemos supor que vamos adicionar à linguagem Java um novo tipo de declaração: uma declaração de propriedade *PropertyDeclaration*. Na versão anterior, precisaríamos definir a própria classe representando a declaração `PropertyDeclaration`, a classe relativa a conjunto simples `PropertyDeclarationSet` e a classe que guardaria todas as declarações de propriedades, `ClassBodyPropertyDeclarationSet`. Além dos métodos *visit* correspondentes a estas três classes, nos três *visitors* atuais (substituição, clonagem e impressão), ou seja, teríamos de definir **3** classes e **9** métodos. Na versão atual, somente a classe original (`PropertyDeclaration`) e os métodos *visit* correspondentes são necessários. Uma redução de **2** classes e **6** métodos para cada tipo de declaração. No que diz respeito à geração automática, significa que menos código precisa ser gerado e, conseqüentemente, menos código precisa ser modificado/mantido.

4.2 GERAÇÃO DE SISTEMAS

Uma vez analisada a arquitetura e identificadas a natureza das classes que a compõem, foi possível definir que pontos podiam ser gerados automaticamente e que pontos deveriam

ser escritos pelo programador.

Podemos intuir que as partes diretamente relacionadas à *linguagem objeto* são aquelas em que o poder de automação é menor. De fato, o módulo de I/O é bastante dependente da *linguagem objeto*, pois é a interface direta com a linguagem, consiste da leitura de um programa na sintaxe da linguagem objeto e da impressão da árvore abstrata em forma textual usando a sintaxe concreta. É possível automatizar boa parte do trabalho, através do uso de transformações em JavaCCTS, que será abordado no próximo capítulo. Neste capítulo vamos nos concentrar nas operações relativas à transformação.

A definição, na linguagem Java, da árvore sintática da *linguagem objeto* é pré-condição para a geração do sistema. Todos os nós que compõem a árvore devem estar definidos. Estes nós precisam conter apenas os atributos relativos a cada nó. Além disto, a árvore deve ser definida de forma unificada. Usando uma superclasse comum a todos os nós. Esta superclasse deve definir atributos comuns a todos os nós.

Para definir as classes que representam os nós, é possível usar ferramentas como o *JJTree* que faz parte da ferramenta JavaCC [Sun06] ou *JTB (Java Tree Builder) [Pur06]*, ambos tomam como entrada um arquivo de especificação de gramática do JavaCC, com algumas anotações, que os direcionam a geração das classes referentes as nós da árvore sintática. Estas ferramentas geram uma classe correspondente a cada não-terminal da gramática, além de outras classes auxiliares. Inclusive, podem gerar a interface do *visitor* para estes nós.

A implementação da árvore é essencial, pois o mecanismo de geração irá modificar esta árvore de entrada, adicionando os métodos necessários para manipulação de transformações. Algumas classes necessárias à manipulação de transformações, por fazerem parte do *framework* reusável de JaTS, não são geradas, mas apenas incorporadas ao sistema final.

A Figura 4.3 ilustra a o funcionamento da abordagem. A árvore sintática da *linguagem objeto*(por exemplo, C# [Lib03]) é fornecida como entrada. Então, geramos boa parte do sistema de transformação para esta linguagem (C#TS), a partir de transformações JaTS. Além disso, algumas partes da implementação de JaTS são reusadas.

De acordo com a arquitetura de JaTS, boa parte do esforço necessário à realização da transformação, está codificado na classe que representa o nó sintático. Desta forma, podemos inferir que o maior esforço da geração consiste na modificação destas classes.

São quatro os passos necessários para a geração destes sistemas: primeiro, a *linguagem objeto* é estendida de forma a se obter a linguagem de *templates*, em seguida, usando-se transformações JaTS, as árvores sintáticas dessas linguagens são adaptadas e os *visitors* são criados. Por fim, os componentes auxiliares, que são compartilhados por todos os sistemas, são adicionados. Estes passos são detalhados nas sub-seções seguintes.

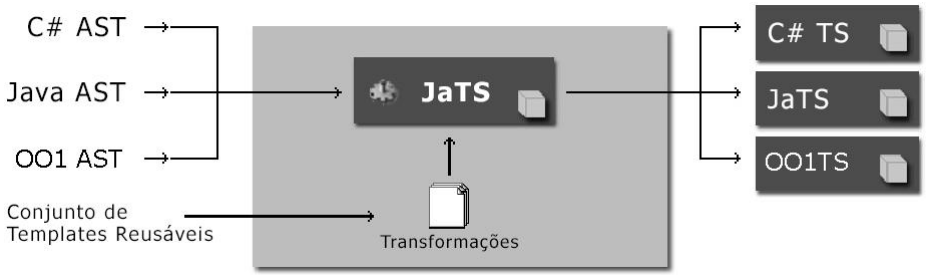

Figura 4.3 Funcionamento da Geração de Sistemas de Transformação

4.2.1 Da Linguagem objeto à Linguagem de Templates

Os sistemas gerados nessa abordagem são baseados em linguagens de *templates*, usadas para definir padrões de programa, cuja sintaxe tenta ser o mais próxima possível da sintaxe da *linguagem objeto*. Isto é feito para diminuir a distância conceitual entre a *linguagem objeto* e a linguagem de codificação das transformações. Permitindo, ao programador, uma mais fácil assimilação do sistema de transformação de programas.

Portanto, a primeira etapa na geração de um novo sistema é criar uma extensão sintática da *linguagem objeto*, que permite a definição de *templates*.

Esta etapa é parcialmente automatizada através do sistema JavaCCTS, que será apresentado no próximo capítulo.

A extensão sintática é realizada em duas estapas:

- Adição de novos *tokens* ao analisador léxico;
- Alteração das produções sintáticas no analisador sintático(*parser*).

4.2.1.1 Analisador Léxico No que se refere a parte léxica da linguagem, precisamos adicionar novas palavras reservadas que representam os tipos da linguagem de *templates*. Esses tipos na linguagem de *templates* correspondem às produções da gramática da *linguagem objeto*. Além disto, precisamos indicar como se forma um identificador de variável e quais os delimitadores para as construções de JaTS:

1. No primeiro passo criamos um *token* `VariableId` que serve para indicar o nome de uma variável. Para isso, primeiro indicamos qual o símbolo de demarcação de variável e depois o associamos ao *token* que determina um Identificador (por exemplo, Identifier). De modo que:

   ```
   AntiQuotation := '#'
   ```

```
VariableId := AntiQuotation Identifier;
```

2. Depois disso, criamos os delimitadores para cláusulas opcionais (em JaTS: "`#[`" e "`]#`") e para expressões executáveis (em JaTS: "`#<`" e "`>#`").

3. Em seguida, adicionamos palavras-chave correspondentes às estruturas sintáticas da *linguagem objeto*. Estas palavras representam os tipos da linguagem de templates. Por exemplo, em `C#TS`, palavras como: `PropertyDeclaration`, `Identifier` e `Type`, são tipos que podem ser usados para qualificar variáveis ou expressões executáveis.

```
PropertyDeclaration:#pd;
Type: #t;
Identifier: #<#att.getName()>#
```

Esta primeira etapa é feita através de transformações em JavaCCTS. A criação de palavras reservadas é feita baseada no nome da produção. Aqui, mostramos um trecho do *template* responsável por criar o *token* referente à variável (`VARIABLE_ID`) e por criar as palavras reservadas da linguagem:

Listing 4.4 Template de criação de tokens

```
// criacao da produção VARIABLE_ID, baseado no token IDENTIFIER
Token:#<Token.createVariableId("VARIABLE_ID","IDENTIFIER","#")>#;

// para cada produção, cria a palavra reservada correspondente
forall #P #in #PDS {

   Token:#<Token.createToken(#<#P.getName()>#)>#;
}
```

4.2.1.2 Analisador Sintático No *parser* as mudanças são maiores, pois são adicionadas novas produções e também são alteradas produções pré-existentes. As produções que são criadas são aquelas que representam construções de JaTS (e.g,`DeclaracaoCondicinal`). As que são alteradas são aquelas produções presentes na *linguagem objeto* que passarão a ter mais derivações para que as construções da linguagem de *templates* sejam suportadas, por exemplo, precisamos mudar `ClassBodyDeclarations` para adicionar a produção `ConditionalDeclaration` como uma de suas expansões.

1. O primeiro passo na extensão da gramática consiste na adição de produções referentes à variáveis e expressões executáveis. Quando falamos em variáveis aqui, não mais estamos falando do *token* que representa uma variável (e.g, `#tmp`), mas estamos falando em permitir que certas construções possam ser representadas por variáveis. Assim, para cada tipo que foi introduzido na especificação léxica da linguagem, que corresponde ao nome de uma produção da gramática, modificaremos a produção correspondente, para introduzir duas novas derivações. Uma delas resultando na produção podendo ser substituída por uma variável, outra em que a produção pode ser substituída pela expressão executável correspondente. Assim, dada uma produção `P`, para qual foi adicionada uma palavra reservada `"P"` será criada uma produção `P'`, tal que `P' := P | PExec() | PVar()`. Por exemplo, para a produção `FieldDeclaration` da gramática de `C#`, criamos a produção `FieldDeclaration'`, onde:

```
FieldDeclaration' := FieldDeclaration  |
                     FieldDeclarationExecutable |
                     FieldDeclarationVariable
FieldDeclarationVariable := "FieldDeclaration" ":" VariableId ";"
FieldDeclarationExecutable :=
                    "FieldDeclaration" ":" ExecutableExpression ";"
```

Onde `ExecutableExpression` é a produção que corresponde às expressões executáveis e que tem a seguinte forma:

```
ExecutableExpression := "#<"ExecInternalExpression">#"
ExecInternalExpression := Expression
```

Esta etapa também pode ser feita via transformações JavaCCTS. O trecho do *template* responsável pela adição dessas novas expansões é o seguinte:

Listing 4.5 Template para adição de expansões

```
//...
forall #P #in #PDS {
   Production:#<#tmp1 = #P ::
     #tmp1.addExpansion(#<#tmp.toExecutableProduction()>#)>#;
   Production:#<#tmp2 = #P ::
```

```
        #tmp2.addExpansion(#<#tmp.toVariableProduction()>#)>#;
}
//...
```

No Capítulo 5, o *template* completo é apresentado, além de um exemplo do código gerado.

Com isto, já é possível termos um sistema de transformação. Pois já existem elementos que permitam casamento, substituição e processamento. Entretanto, apenas com variáveis e expressões executáveis, o poder de expressividade da nova linguagem de *templates* seria limitado. Assim, para flexibilizar a escrita de *templates* e dar maior poder de decisão à geração de código e à transformação de programas, como em JaTS, adicionamos cláusulas opcionais e outras declarações executáveis, para iteração e controle de fluxo. Os passos a seguir são feitos manualmente.

2. A adição de cláusulas opcionais é um passo que não apenas depende da sintaxe da linguagem, mas também de semântica. Pois se as cláusulas opcionais fossem adicionadas de forma indiscriminada, o resultado das transformações poderiam ser programas inválidos na *linguagem objeto*.

   Portanto, neste ponto, identificamos elementos que podem ser opcionais na *linguagem objeto* e então modificamos a estrutura das produções que às contém para que os aceite como uma estrutura opcional. Em C#, assim como em Java, quando uma classe não estende nenhuma outra, ela, implicitamente, estende a classe `Object`. Assim, podemos modificar a produção que corresponde a uma declaração de classe em C# (`ClassDeclaration`), para que a derivação correspondente a superclasse seja opcional:

   ```
   ClassDeclaration :=  Modifiers "class" Identifier [ ":" NameList ]
   ClassBody
   ```

   Mudaria para:

   ```
   ClassDeclaration :=  Modifiers "class" Identifier [ (":" NameList) |
    ("#[" ":"  NameList "]#") ] ClassBody
   ```

3. Por fim, adicionamos as produções para iteração e para transformação condicional(declarações condicionais e declarações iterativas). Estas produções devem ser adicionadas no mesmo nível das demais declarações presentes na linguagem. No caso de C#, seria no mesmo nível de declarações de método e de propriedades.

   Uma produção para iteração tem a forma:

```
IterativeDeclaration := "forall" VariableId "in"
                  (VariableId |ExecutableExpression)
                  "{"  Declaration  "}"
```

   A produção referente a declaração condicional é bastante similar a um bloco *if-then-else*:

```
ConditionalDeclaration := "#if" (<Expression>) "{"
                                  Declaration
                          "}" ["else" "{" Declaration "}"]
```

Quando aplicável, pode-se adicionar construções de metaprogramação que permitem realizar o casamento de conjunto de declarações. Em JaTS-TL, temos a produção `FieldDeclarationSet`, por exemplo.

Esta tipo de construção é a base para a definição de um casamento de padrões extremamente poderoso, pois é através de variáveis que representam conjuntos de declarações que podemos ter um casamento de padrões para declarações que ignora a ordem em que estas são declaradas e ignora também se estas se apresentam intercaladas com outras declarações.

Na abordagem adotada em MetaJ [Oli04], os passos referentes a adição de novas produções representando variáveis e a introdução de cláusulas opcionais podem ser automatizados através do processador de gramáticas Cup [Hud06].

4.2.2 Transformação da Árvore Sintática

Segundo foi visto na Seção 4.1, é com os nós que fica a maior parte da responsabilidade pela transformação. Portanto, o maior esforço na geração de um sistema de transformação baseado em JaTS é na transformação das classes que representam os nós sintáticos.

Os métodos responsáveis pelo casamento e processamento são implementados em cada nó, seguindo o padrão *Interpreter*. O nó também tem o método para receber os *Visitors* responsáveis pela impressão da árvore e pela substituição de valores. Estes métodos são gerados através de transformações JaTS. É importante ressaltar que as transformações

aqui apresentadas, fazem parte de um conjunto de *templates* de JaTS, porém, durante a implementação propriamente dita, é possível modificar estes para que fiquem o mais próximo possível do código real. Por exemplo, mudando o nome de uma determinada classe ou de um pacote.

Após a refatoração, restou apenas uma classe de nós que precisa ser gerada. As demais podem ser simples instanciações das classes genéricas que fazem parte do *framework* de JaTS.

Assim, usamos apenas uma transformação e a aplicamos a todos os nós da árvore. Para cada nó, inicialmente, são capturadas as informações de interesse. Interessa-nos, especialmente, o conjunto de declarações de atributos desta classe que não pode estar vazio, pois é a partir dos atributos que as operações são determinadas. As demais informações dos nós sintáticos são apenas conservadas.

O *template* a seguir é o lado esquerdo da transformação e é usado para capturar as informações da classe de entrada:

Listing 4.6 Template de casamento para os nós da árvore

```
ModifierList:#M class #TYPE #[ extends #TYPE_SC ]#
                            #[ implements #TYPE_IL ]#{

    FieldDeclarationSet:#TYPE_ATTRS;
    ConstructorDeclarationSet:#TYPE_CDS;
    MethodDeclarationSet:#TYPE_MDS;
    ...
}
```

O template é definido de forma bem genérica, para poder casar com qualquer classe.

Em seguida, usamos o *template* de geração para gerar os métodos relativos à transformação. Opcionalmente, podem ser geradas estruturas auxiliares, como métodos de acesso aos atributos, construtores, método *equals()*. O *template* a seguir mostra como o método para casamento é gerado. Os métodos *process* e *accept* não serão mostrados, para facilitar a leitura. Porém, estão completamente descritos no Apêndice A.

Listing 4.7 Template de geração do método match

```
ModifierList:#M class #TYPE #[ extends #TYPE_SC ]#
                              #[implements INode, #TYPE_IL ]# {
```

```
  FieldDeclarationSet:#TYPE_ATTRS;
  ...
  public void match(INode node, ResultSet results)
            throws NodesNotMatchedException {
    ...
    ResultSet rs = (ResultSet) results.clone();
    if (!this.matchesAsAVariable(node, rs, INode.NO_TYPE)){
      let #varName = #<#TYPE.toVariableName()>#;
      in
        #TYPE  #varName = (#TYPE) node;
        forall #A #in #TYPE_ATTRS {
          let #attName = #<#A.getName()>#;
              #varType = #<#A.getType()>#;
          in
            #if (#varType instanceof INode) {
              this.#attName.match(#varName.#attName, rs);
            }
            else {
              #if (#varType.isPrimitive()){
                if (!(this.#attName == #varName.#attName)) {
                  throw new NodesNotMatchedException();
                }
              }
              else {
                if (!this.#attName.equals(#varName.#attName)) {
                  throw new NodesNotMatchedException();
                }
              }
            }
        }
    }
    results.merge(rs);
  }
  // preservação dos construtores e métodos
  ConstructorDeclarationSet:#TYPE_CDS;
  MethodDeclarationSet:#TYPE_MDS;

}
```

Esta transformação é aplicada a uma classe representando um nó da árvore sintática, como o da Listagem 4.8. O *template* da Listagem 4.6 é casado com este nó, fazendo com que a variável `FieldDeclarationSet:#TYPE_ATTRS;` armazene todos os atributos classe. Estes atributos serão usados pelo *template* da Listagem 4.7 para a geração do método *match* e de outros métodos não listados aqui (*process* e *accept*).

Para ilustrar o que esta transformação faz, veremos um exemplo de classe de entrada e a saída produzida por esta transformação. A classe a seguir, faz parte da implementação da AST (Abstract Syntax Tree) de C# e representa uma expressão binária:

Listing 4.8 Classe BinaryExpression antes da transformação

```
public class BinaryExpression {
    private Expression leftOperand;
    private int operator;
    private Expression rightOperand;

    public BinaryExpression(){ }

    public boolean equals(Object obj){ }

}
```

Após a aplicação da transformação supracitada, ela ficará assim:

Listing 4.9 Classe BinaryExpression transformada

```
public class BinaryExpression implements INode {

  private Expression leftOperand;
  private int operator;
  private Expression rightOperand;

  public BinaryExpression(){ }
  public boolean equals(Object obj){ }

  public void match(INode node, ResultSet results)
           throws NodesNotMatchedException {
    ResultSet rs = (ResultSet) results.clone();
```

```
    if (!this.matchesAsAVariable(node,rs,INode.NO_TYPE)) {
      BinaryExpression binaryExpression = (BinaryExpression) node;
      this.leftOperand.match(binaryExpression.leftOperand, rs);
      if (!(this.operator == binaryExpression.operator)) {
        throw new NodesNotMatchedException();
      }
      this.rightOperand.match(binaryExpression.rightOperand, rs);
    }
    results.merge(rs);
  }
}
```

Para cada atributo presente na classe, foi gerada uma entrada no método *match*. É importante notar que no caso do atributo `operator`, não foi gerada uma chamada ao método *match*, pois este atributo possui um tipo primitivo, o que geraria um programa inválido. Se existisse um outro atributo de um tipo Java qualquer que não implementasse a interface `INode` também não seria gerada uma chamada ao método *match*, mas sim, seria feita uma comparação usando o método *equals*. Outras verificações também são usadas no *template* completo, porém, simplificamos o *template* apresentado, para facilitar o entendimento por parte do leitor.

A seguir, o programa, escrito em JaTS-ML, que aplica as transformações em todos os nós, presentes em um diretório `"fontes\nodes"`:

Listing 4.10 Programa em JaTS-ML para transformação dos nós

```
LoadTemplate("lhs.jats");

LoadTemplate("rhs.jats");

Transformation t <- #left :=> #right;

TypeDeclarationSet #fontes <- parse("fontes/nodes");

forall #f in #fontes {
    ResultSet res;
    Code result <- #f < t/res;
    print(result, #f.getFileName());
}
```

4.2.3 Geração de Visitors

JaTS utiliza-se de *Visitors* não apenas para implementar a operação de substituição de valores, mas também para implementar a clonagem dos nós da árvore e a impressão. Diferente do que acontece com a transformação de nós, a transformação de geração de *visitors* é aplicada `n` vezes (onde `n` é a quantidade de classes que representam nós da árvore sintática) e gera uma única classe de saída. A cada aplicação da transformação, é adicionado um método *visit* correspondente à classe do nó sintático corrente.

Na definição desta transformação, usamos três *templates*: um para capturar informações do nó sintático para o qual será gerado o método *visit*, um para capturar as informações do *visitor* que está sendo transformado e um *template* para a geração do método *visit* e conservação dos métodos previamente existentes.

Os templates para capturar informação do nó sintático e do *visitor* são similares ao apresentado na Listagem 4.6, por isso serão omitidos. A seguir mostramos o template de geração do *visitor*:

Listing 4.11 Template de geração do método visit, do *ReplacementVisitor*.

```
public class #REPLACEMENT_VISITOR #[extends #SC_VISITOR]#
                                  #[implements #IVISITOR]# {
  FieldDeclarationSet:#V_FDS;
  ConstructorDeclarationSet:#V_CDS;
  MethodDeclarationSet:#V_MDS;

  public Object visit(#TYPE node, Object data){
    if (node == null) {
      throw new IllegalArgumentException();
    }
    forall #A #in #TYPE_ATTRS {
      let #tmpVar  = #<#A.addPrefix("tmp")>#;
          #varType = #<#A.getType()>#;
            in
          #varType #tmpVar = node.#<#A.addPrefix("get")>#();
          #if(#varType instanceOf INode) {
            if (#tmpVar.isVariable()) {
              node.#<#A.addPrefix("set")># (
                  (#<#A.getType()>#)ReplacementHelper.getValueOf(#tmpVar));
            }
          }
```

```
      return node;
    }
  }
}
```

Esta transformação é aplicada para cada nó da árvore sintática e irá gerar tantos métodos *visit* quantos forem os nós da árvore. Vamos ilustrar a aplicação desta transformação ao *visitor* descrito na Listagem 4.12 em conjunto com a classe apresentada na Listagem 4.13.

Listing 4.12 Classe ReplacementVisitor antes da transformação

```
public class ReplacementVisitor implements IVisitor {
  ...
  public Object visit(Identifier node, Object data){ }
  public Object visit(UnaryExpression node, Object data){ }
}
```

A transformação produz como saída a classe `ReplacementVisitor` contendo um novo método *visit*:

Listing 4.13 Classe ReplacementVisitor transformada

```
public class ReplacementVisitor implements IVisitor {
  ...
  public Object visit(Identifier node, Object data){ }
  public Object visit(UnaryExpression node, Object data){ }

  public Object visit(BinaryExpression node, Object data){
    if (node == null) {
      throw new IllegalArgumentException();
    }
    Expression tmpLeftOperand = node.getLeftOperand();
    if (tmpLeftOperand.isVariable()) {
      node.setLeftOperand(
           (Expression)ReplacementHelper.getValueOf(tmpLeftOperand));
    }
    Expression tmpRightOperand = node.getRightOperand();
```

```
    if (rightHand.isVariable()) {
      node.setRightOperand(
           (Expression)ReplacementHelper.getValueOf(tmpRightOperand));
    }
    return node;
  }
}
```

De forma similar, existem templates para geração do método *visit* correspondente a cada nó dentro dos *visitors* de clonagem e de impressão. O *visitor* de impressão não é gerado por completo, uma vez que a partir da sintaxe abstrata (representada pelas classes de entrada do sistema) não é possível obter a sintaxe concreta. Então, o programador precisa inserir o código que monta a `String` correspondente ao nó sintático.

Uma opção para resolver este problema, seria usar anotações na gramática, permitindo que se pudesse extrair informação a respeito da sintaxe concreta. Estas anotações poderiam ser descritas usando o recurso de anotações de Java 1.5.

4.2.4 Componentes Auxiliares

A refatoração realizada no código do JaTS, permitiu que mais componentes do núcleo da ferramenta fossem reaproveitados para os demais sistemas. A estruturação foi feita de forma que tais nós somente dependessem da interface `INode`. Sendo assim, para completar a geração do sistema, adicionam-se estas classes ao conjunto de classes gerados:

ParserHelper auxilia na criação de nós da árvore sintática.

TransformHelper auxilia no processamento dos nós. Realiza a troca de um elemento pelo resultado de seu processamento.

ReplacementHelper auxilia no processo de substituição de alguns nós, pois JaTS permite que alguns nós sejam intercambiáveis entre si. Por exemplo: Uma variável do tipo `Identifier` pode ser substituída por objeto do tipo `Name` que só contém um identificador.

ResultSet mantém um conjunto de mapeamentos variável-valor usado por todas as classes que dizem respeito à transformação.

EvaluationFunctions Contém uma série de métodos que auxiliam na avaliação de uma expressão. Ex: método para avaliar uma expressão binária, como `"tmp"+2`, que resulta na String `"temp2"`.

ExecutableExpression Classe que representa uma expressão executável. Contém a lógica necessária para a execução da expressão.

ConditionalDeclaration Classe que representa uma declaração condicional.

IterativeDeclaration Classe que representa uma declaração iterativa.

NodeList <T extends INode> Representa uma lista de nós, cuja ordem de declaração importa.

GenericDeclarationSet<T extends Declaration> Representa um conjunto de nós. A ordem das declarações não importa. As declarações que podem ser usadas para parametrização, devem, necessariamente, implementar a interface Declaration.

GenericBodyDeclarationSet<T extends Declaration> Representa o conjunto de declarações contidas num corpo de uma declaração de tipo.

Além disto, os pacotes abaixo descritos são incorporados ao sistema final.

cin.jats.engine Pacote com as classes responsáveis por orquestar a transformação.

cin.jats.engine.util Pacote contendo classes utilitárias, como `NodeStack` usada para a operação de clonagem, `JaTSHashTable`, usada para implementar o `ResultSet`

cin.jats.engine.nodes.exception Exceções relativas aos nós.

4.2.5 Processo de Desenvolvimento

Esta seção tenta resumir os passos que foram descritos nas seções anteriores de modo a ilustrar como estes passos estão distribuídos no tempo.

A Figura 4.4 traz o processo de desenvolvimento usando esta abordagem. Destacando que passos são executados pelo desenvolvedor do sistema de transformação e que passos são automatizados pelo JaTS.

A seguir as atividades serão descritas de acordo com o seguinte template:

Recurso Indica se a ativade é executada pelo manualmente pelo desenvolvedor, ou se auxiliada pelo JaTS ou pelo JavaCCTS

Entrada Artefatos de entrada

Saída Artefatos de saída

Passos Passos para execução da tarefa

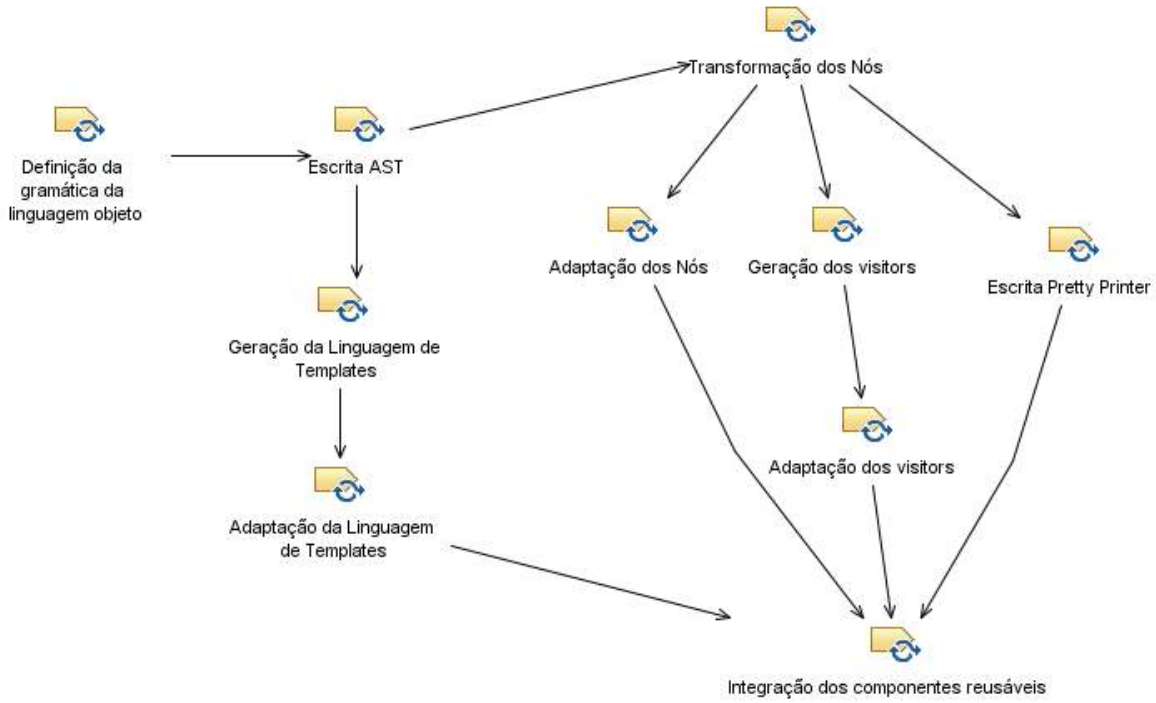

Figura 4.4 Processo de Geração de um Sistema de Transformação

4.2.5.1 Definição da gramática da linguagem objeto

Recurso Desenvolvedor

Entrada

Saída Gramática da linguagem

Passos Definir os não-terminais e as produções da gramática

4.2.5.2 Escrita AST

Recurso Desenvolvedor

Entrada Gramática da linguagem

Saída Árvore sintática da linguagem

Passos

- Criar, para cada não-terminal da gramática, uma classe Java correspondente

- Cada classe, deve conter apenas os atributos necessários para representar o nó da árvore sintática

4.2.5.3 Geração da linguagem de templates

Recurso JavaCCTS

Entrada Gramática da linguagem objeto em JavaCC

Saída Gramática da linguagem de transformação em JavaCC.

Passos

- Executar a transformação para adição de novos tokens e de novas produções no arquivo de gramática da linguagem objeto.

4.2.5.4 Adaptação da linguagem de templates

Recurso Desenvolvedor

Entrada Gramática da linguagem de transformação

Saída Gramática da linguagem de transformação modificada

Passos

- Verificar se o parser gerado pela JavaCC está correto
- Executar modificações necessárias

4.2.5.5 Transformação dos nós

Recurso JaTS

Entrada

- Classes correspondentes aos nós da AST
- Templates para modificação dos nós

Saída Classes correspondentes aos nós da AST modificadas, com métodos para transformação

Passos

- Para cada classe correspondente a um nó da AST, executar a transformação de geração de métodos responsáveis por realizar as transformações.

4.2.5.6 Adaptação dos nós

Recurso Desenvolvedor

Entrada Classes correspondentes aos nós da AST

Saída Classes correspondentes aos nós da AST modificadas

Passos

- Para cada classe referente a um nó da AST, adicionar os métodos que forem convenientes para facilitar a escrita de transformações. Métodos que poderão ser chamados via declarações executáveis.

4.2.5.7 Geração dos visitors

Recurso JaTS

Entrada

- Classes correspondentes aos nós da AST
- Templates para geração dos visitors

Saída Classes correspondentes aos visitors.

Passos

- Aplicar a transformação de geração de métodos visit para cada visitor. A transformação deve ser aplicada tantas vezes quantos forem os nós da árvore sintática.

4.2.5.8 Adaptação dos visitors

Recurso Desenvolvedor

Entrada Classes correpondentes aos visitors

Saída Classes correspondentes aos visitors modificadas

Passos

- Verificar eventuais problemas nas classes correpondentes aos visitors
- Corrigir problemas

4.2.5.9 Escrita do PrettyPrinter

Recurso Desenvolvedor, JaTS

Entrada

- Classes referentes aos nós da AST
- Gramática da linguagem
- Transformação basica para o Pretty

Saída PrettyPrinterVisitor

Passos

- Gerar o PrettyPrinterVisitor usando a transformação correspondente
- Adaptar todos os métodos *visit* de acordo com a gramática da linguagem, fazendo com que o texto gerado seja correpondente a gramática concreta da linguágem.

4.2.5.10 Integração dos componentes reusáveis

Recurso Desenvolvedor

Entrada

- árvore sintática modificada
- visitors modificados
- pretty printer
- componentes reusáveis de JaTS

Saída Sistema de transformação

Passos

- Compilar todos os arquivos e verificar se existe alguma pendência
- Corrigir eventuais problemas

CAPÍTULO 5

UM SISTEMA DE TRANSFORMAÇÕES PARA A GRAMÁTICA DO JAVACC

Motivados pela necessidade de automatizar passos da criação de uma linguagem de *templates* a partir da definição da *linguagem objeto*, decidimos usar a abordagem para geração de sistemas de transformação proposta neste trabalho para criar um sistema de transformação para a linguagem de especificação de gramáticas da ferramenta JavaCC [Sun06], o qual denominamos JavaCCTS (*JavaCC Transformation System*).

Por ser um gerador de *parsers* para a linguagem Java, o JavaCC possui uma gramática que compartilha várias produções com a linguagem Java. De fato, a linguagem para definição de gramáticas do JavaCC é uma extensão sintática da linguagem Java com construções que permitem a especificação de expressões regulares e produções gramaticais. Assim, usamos a implementação de JaTS como um subconjunto da implementação do sistema de transformação para JavaCC.

Como foi criado com o objetivo explícito de automatizar a tarefa de criação de um parser para a linguagem de *templates*, nem todas as construções do JavaCC são passíveis de serem transformadas. Para aquelas seções da especificação da gramática em que não havia necessidade de modificação, apenas replicamos o código original, tal qual ele se apresenta na gramática de origem. Este é o caso, por exemplo, da primeira seção do arquivo, que contém opções de geração do *parser*.

Antes de prosseguirmos, convém mostrarmos como um arquivo de especificação de gramáticas JavaCC é definido:

Listing 5.1 Gramática JavaCC

```
javacc_input ::= javacc_options
  "PARSER_BEGIN" "(" <IDENTIFIER> ")"
  java_compilation_unit
  "PARSER_END" "(" <IDENTIFIER> ")"
  ( production )*
  <EOF>
```

A classe que representa `java_compilation_unit` é exatamente a mesma usada em JaTS. Porém, para efeitos da criação da linguagem de *templates* para JavaCC, esta classe não é relevante e é apenas copiada no código final. No sistema JavaCCTS criamos uma variável para representar o código Java, cujo tipo é `JavaCompilationUnit`.

Num arquivo JavaCC, a especificação léxica e sintática, são definidas como produções (`production`):

Listing 5.2 Produções no JavaCC

```
production ::= javacode_production
  regular_expr_production
  bnf_production
  token_manager_decls
```

Para nós, interessam as produções referentes aos componentes léxicos e às produções definidas usando a notação de BNF[1](`regular_expr_production` e `bnf_production`, respecitivamente).

A produção que armazena componentes léxicos é dividida em 4 seções, mas nos interessa apenas uma:`TOKEN`, onde adicionamos palavras reservadas e delimitadores. As demais seções são apenas replicadas no código final.

A produção que usa a notação de BNF é a mais usada dentro da especificação do JavaCC, pois é através dela que se definem a maioria das produções de uma gramática qualquer. Esta produção tem a seguinte forma:

Listing 5.3 Gramática para cada produção

```
bnf_production ::= java_access_modifier java_return_type
java_identifier "(" java_parameter_list ")" ":"
  java_block
  "{" expansion_choices "}"

expansion_choices ::= expansion ( "" expansion )*

expansion ::= ( expansion_unit )*

expansion_unit ::= local_lookahead
  java_block
```

[1]Backus-Naur Form: Notação formal para descrever a sintaxe de uma linguagem. Usada, em especial, para linguagens computacionais

```
  "(" expansion_choices ")" [ "+" | "*" | "?" ]
  "[" expansion_choices "]"
  [ java_assignment_lhs "=" ] regular_expression
  [ java_assignment_lhs "=" ] java_identifier "(" java_expression_list ")"
```

Uma `bnf_production` tem elementos muito parecidos com uma definição de método em Java, com adição da parte referente às produções propriamente ditas `expansion_choices`. Esta é a produção da gramática de JavaCC em que precisamos fazer a maioria das alterações. Na Seção 4.2.1.2 mostramos com uma produção é modificada, de forma a permitir variáveis e expressões executáveis. Vamos voltar ao exemplo mostrado nela e, em seguida, mostramos como ele seria executado no JavaCCTS.

Se tivéssemos uma produção qualquer `FieldDeclaration`, o resultado seria o seguinte:

```
FieldDeclaration' := FieldDeclaration  |
                     FieldDeclarationExecutable |
                     FieldDeclarationVariable
FieldDeclarationVariable := "FieldDeclaration" ":" VariableId ";"
FieldDeclarationExecutable :=
                    "FieldDeclaration" ":" ExecutableExpression ";"
```

Usando a notação do JavaCC e supondo a existência da produção `FieldDeclaration`:

Listing 5.4 Exemplo de produção

```
FieldDeclaration():{} {
   [Modifier()] Type() Identifier() ["=" Initializer() ] ";"
}
```

A produção FieldDeclaration original conta com apenas uma (`expansion_unit`). Durante a transformação, adicionamos outras duas:

Listing 5.5 Exemplo de produção resultante

```
FieldDeclaration():{} {
```

```
  (
    LOOKAHEAD("FieldDeclaration" : VariableId())
    "FieldDeclaration" : VariableId()

    LOOKAHEAD("FieldDeclaration" : ExecutableDeclaration())
    "FieldDeclaration" : VariableId()

   [Modifier()] Type() Identifier() ["=" Initializer() ] ";"
  )
}
```

A adição é sempre feita no início, para evitar problemas no *parser* gerado. Além disso, usamos o mecanismo de *lookAhead* para garantir que a gramática gerada não terá ambiguidades.

5.1 ELEMENTOS DE JAVACCTS

Para poder criar uma arquivo JavaCC, com as alterações necessárias à criação da linguagem de templates, definimos os seguintes tipos em JavaCCTS:

Token Representa uma entrada na seção `Tokens` do JavaCC.

TokenSet Representa o conjunto de *tokens* do JavaCC.

Skip Representa uma entrada na seção `Skip` do JavaCC (é apenas replicada na saída).

More Representa uma entrada na seção `More` do JavaCC (é apenas replicada na saída).

SpecialToken Representa uma entrada na seção `More` do JavaCC (é apenas replicada na saída).

ProductionSet Armazena todas as produções presentes na especificação da gramática.

Production Representa um não-terminal da gramática.

ExpansionChoice Representa o conjunto de expansoes expansões possíveis para uma produção.

ExpansionUnit Representa cada uma das expansões presente na produção.

Para representar a árvore do JavaCC, todos os nós de JaTS foram reusados e algumas classes foram criadas para representar os tipos supracitados. Estas classes foram transformadas usando as transformações JaTS-TL do Capítulo 4.

A Figura 5.1 ilustra as principais classes do JavaCCTS.

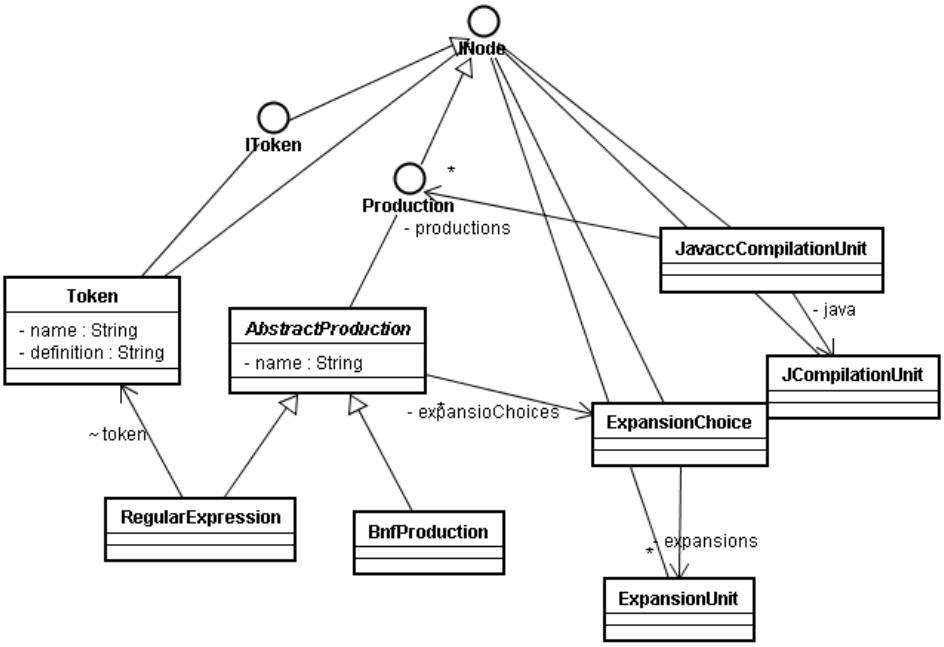

Figura 5.1 Classes do JavaCCTS

5.2 TRANSFORMANDO UMA GRAMÁTICA USANDO JAVACCTS

Entre os passos necessários para a criação de uma linguagem de templates citados na Seção 4.2.1, o JavaCCTS suporta aqueles referentes à parte léxica da linguagem e á definição de produções referentes a variáveis e expressões executáveis. As transformações possíveis são:

- Adicionar novas palavras reservadas (”FieldDeclaration”, ”Type”)
- Adicionar a produção para variável (VariableId)
- Adicionar a produção para expressões executáveis (ExecutableExpression)
- Adicionar uma expansão para tratamento de variáveis (Por exemplo, ”FieldDeclaration:”VariableId)

- Adicionar uma expansão para tratamento de expressões executáveis (Por exemplo, "FieldDeclaration:"ExecutableExpression)

A transformação é composta por um *template* de casamento e um *template* de geração e é aplicada a um arquivo de especificação de gramáticas do JavaCC.

O seguinte *template* é usado para capturar informações do JavaCC:

Listing 5.6 Template de casamento do JavaCC

```
// irá preservar as opcoes
Options:#javacc_options;
// irá preservar o código java
JavaCompilationUnit:#java_code;
// irá preservar a seção skip
Skip:#skip_section;
// irá armazenar todos os tokens, poderá ser manipulado
TokenSet:#tokens;
//armazena todas as produções. Será manipulado
ProductionSet:#PDS;
```

E o seguinte template irá adicionar as palavras reservadas, as expansões das produções e duas novas produções:

Listing 5.7 Adição de produções para variável e para expressão executável

```
Options:#javacc_options;

 JavaCompilationUnit:#java_code;

Skip:#skip_section;
// mantém os tokens originais
TokenSet:#tokens;

// cria o token referente a variável, indicando qual é
// o token que servirá de base (IDENTIFIER)
Token:#<#Token.createVariableId("VARIABLE_ID","IDENTIFIER","#")>#;

// adiciona a producao para variavel, a partir
```

```
Production:#<#RegExpProduction.createVariableId("VARIABLE_ID")>#;

// adiciona a producao para expressao executavel, baseada
// na producao indicada (Expression) e nos delimitadores
// ExecutableExpression := "#<" Expression() ">#"
Production:#<#BnfProduction.createExecExpProduction(
                            "#<",">#","Expression")>#;

// preserva as procuções atuais
ProductionSet:#PDS;

// para cada produção P, cria a palavra chave correspondente,
// cria as producões P_Variable e P_Executable
// e as adiciona como filhos de P
forall #P #in #PDS {

   Token:#<Token.createToken(#<#P.getName()>#)>#;
   Production:#<#tmp1 = #P ::
     #tmp1.addExpansion(#<#tmp.toExecutableProduction()>#)>#;
   Production:#<#tmp2 = #P ::
     #tmp2.addExpansion(#<#tmp.toVariableProduction()>#)>#;
}
```

Para ilustrar o que esta transformação faz, consideremos este exemplo de gramática para expressões aritméticas simples:

Listing 5.8 Especificação da gramática antes da transformação

```
PARSER_BEGIN(AritmeticGrammar) class AritmeticGrammar {
  public void test(){}
} PARSER_END(AritmeticGrammar)


TOKEN : /* IDENTIFIERS */ {
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
}

SimpleNode Start() : {} {
  Expression() ";"
```

```
}

void Expression() : {} {
  AdditiveExpression()
}

void AdditiveExpression() : {} {
  UnaryExpression()
  ( ( "+" | "-" ) UnaryExpression() )*
}

void UnaryExpression() : {} {
  "(" Expression() ")" | Identifier() | Integer()
}
```

A gramática resultante tem todas as suas produções alteradas, para a adição de novas expansões. Além disso, duas produções foram criadas: `VariableId()` e `ExecutableExpression()`.

Listing 5.9 Especificação da gramática depois da transformação

```
PARSER_BEGIN(AritmeticGrammar) class AritmeticGrammar {
  public void teste(){}
} PARSER_END(AritmeticGrammar)

SKIP : {
  " "
| "\t" | "\n" | "\r" }

TOKEN : /* IDENTIFIERS */ {
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
}

// gerado
TOKEN: {
  < VARIABLE_ID: "#"<IDENTIFIER> >
}
TOKEN: {
  < Start: "Start" >
  |
```

```
  < Expression: "Expression" >
  |
  < UnaryExpression: "UnaryExpression" >
  |
  < Identifier: "Identifier" >
}

SimpleNode Start() : {} {
 ( LOOKAHEAD("Start" ":" VariableId())
  "Start" ":" VariableId()
  |
  LOOKAHEAD("Start" ":" ExecutableExpression())
  "Start" ":" VariableId()
  |
   Expression() ";"
  )
}


void Expression() : {} {

( LOOKAHEAD("Expression" ":" VariableId())
  "Expression" ":" VariableId()
  |
  LOOKAHEAD("Expression" ":" ExecutableExpression())
  "Expression" ":" VariableId()
  |
  AdditiveExpression()
 )
}
...

void UnaryExpression() : {} {
 ( LOOKAHEAD("UnaryExpression" ":" VariableId())
  "UnaryExpression" ":" VariableId()
  |
  LOOKAHEAD("UnaryExpression" ":" ExecutableExpression())
  "UnaryExpression" ":" VariableId()
  |
```

```
  "(" Expression() ")" | Identifier() | Integer()
  )
}

// producao nova
void VariableId(): {} {
   <VARIABLE_ID>
}
// producao nova
void ExecutableExpression() : {} {
   "#<" Expression() ">#"
}
```

Como vimos, automatizamos boa parte do trabalho de criar uma linguagem de *templates* baseada numa *linguagem objeto*. Ainda não implementamos os dois últimos pontos da criação desta lingaugem, conforme citado na Seção 4.2.1.2, porém, com o que já foi implementado, o custo da escrita de uma linguagem de *templates* cai drasticamente, pois deixa de ser proporcional ao tamanho da BNF e passa a ser o custo de adicionar duas novas produções e ajustar o *parser* gerado.

CAPÍTULO 6

AVALIAÇÃO

Para atestar que a abordagem definida no Capítulo 4 tem aplicabilidade, realizamos alguns testes. Em todos, pudemos perceber que a quantidade de código gerada é realmente representativa dentro do código necessário a um sistema de transformação de programas real. Além disto, motivados pelos resultados de alguns testes, decidimos criar um novo sistema o JavaCCTS, como visto no Capítulo 5 para aumentar a quantidade de código gerado automaticamente e, assim, permitir que um novo sistema de transformação seja facilmente criado, quando necessário.

Inicialmente, testamos a abordagem com o próprio JaTS, para permitir a evolução da ferramenta. Isso é um teste bastante importante e bastante freqüente no âmbito de sistemas de transformação: usar o sistema de transformação para ajudar a desenvolver ele mesmo. Técnica conhecida como *bootstrapping*.

Além da evolução de JaTS, usamos a abordagem em outras três ocasiões: para criar sistemas de transformação para as linguagens C# [Lib03], OO1 [CIn06] e, por fim, para a linguagem de especificação de gramáticas da ferramenta JavaCC.

As seções seguintes detalham cada uma das experiências com a abordagem.

6.1 BOOTSTRAPPING DE JATS

O primeiro uso desta abordagem foi para realizar o *bootstrapping* de JaTS. No momento em que este trabalho começou, certas construções de Java ainda não eram completamente suportadas por JaTS. Os comandos (“`If`”, “`While`”, “`For`”, etc), por exemplo, não possuíam suporte a casamento, apenas eram mantidos se existissem no programa Java original. Então, esta abordagem foi empregada para gerar as classes que representavam tais comandos e também os métodos referentes nos *visitors* usados na implementação JaTS. Além disto, a abordagem também foi usada para introduzir o suporte a *Inner Classes* e inicializadores.

A linguagem Java também passou por alterações e esta abordagem foi usada para dar suporte as novas construções da linguagem. Como se pode supor, neste caso, o código gerado correspondia a quase totalidade do código de produção. Houve alterações no *parser*, porém apenas para colocar as ações semânticas correspondentes à criação dos nós

da árvore e para acrescentar as novas construções da linguagem.

O código final só não é completamente gerado automaticamente, porque em JaTS temos o que chamamos de "*métodos de API*", métodos que atuam sobre os nós de árvore sintática, porém, cuja interface de entrada é restrita aos tipos primitivos da linguagem Java e *String*. O objetivo de criar tais métodos é facilitar o trabalho do programador que codifica as transformações, permitindo que este escreva de forma mais simples e intuitiva, código de manipulação da árvore sintática. Por exemplo, se na classe correspondente a uma declaração de método `MethodDeclaration`, é gerado um método `setExceptionList(NameList exceptionList)` e o programador que codifica as transformações precisa adicionar uma exceção a um determinado método, ele escreveria algo assim:

```
#<#md.setExceptionList(#md.getExceptionList().add(new
Name("SystemException")))>#;
```

Porém, para facilitar a escrita, criamos um método na classe `MethodDeclaration` que torna esta operação muito mais intuitiva, pois é ele quem encapsula o trabalho que seria do programador:

Listing 6.1 Classe MethodDeclaration

```
public class MethodDeclaration {

    private NameList exceptionList;

    public void addException(String exception) {
        this.exceptionList.add(new Name(exception));
    }
}
```

Assim, digamos que exista uma variável `MethodDeclaration:#md;` na qual queremos executar uma operação, adicionando uma nova exceção `SystemException`, podemos escrever simplemente:

```
#<#md.addException("SystemException")>#;
```

É importante ressaltar que tal recurso serve somente para melhorar a legibilidade das transformações escritas em JaTS. Pois processamento já seria suportado pela implementação gerada automaticamente.

No que concerne a atualização da sintaxe de Java, duas construções foram adicionadas por meio desta abordagem.

- Suporte a enumerações
- Suporte a anotações

Como esperado, este é o teste que obteve os melhores resultados, já que pouco código precisou ser inserido manualmente: o código da árvore sintática foi gerado automaticamente, com um percentual de aproximadamente 92%. O código dos *visitors* também foi alterado automaticamente e pouco esforço foi necessário para a alteração do *parser*, pois somente foi preciso adicionar o referente às novas construções, já que produções para variáveis e declarações executáveis já estavam presentes.

6.2 C#TS: UM SISTEMA DE TRANSFORMAÇÃO PARA C#

Tão logo as transformações se mostraram viáveis para a extensão do JaTS para suportar novas características de Java, decidimos usar a mesma abordagem na geração de um sistema de transformações para a linguagem C# [Lib03].

A linguagem C# guarda muitas similaridades sintáticas e semânticas com a linguagem Java. Destacando-se por apresentar mais construções do que esta última, como:

Namespaces Similares a declarações de pacotes em Java, porém, além de encapsular os tipos declarados, são aninháveis entre si.

Enums Enumerações. Representam tipos definidos através de um numero finito de valores. Foi incorporado na versão 1.5 da linguagem Java.

Structs Definem tipos de forma similar ao que acontece com classes, entretanto, tipos *struct* são sempre passados por valor.

Delegates Definem uma espécie de *ponteiro* para um método em C#.

Events Eventos são membros que permitem a um objeto ou a uma classe prover notificação para outros. Estão diretamente associados a *delegates*

Properties Propriedades servem para acessar atributos. Uma propriedade é uma notação simplificada para os métodos de acesso a um atributo (escrita e leitura)

Indexers Indexadores são propriedades especiais que permitem navegar pelos atributos de uma classe, usando uma notação bastante similar a de um acesso a array.

Attributes Atributos são metadados que podem ser úteis na geração de código em tempo de execução.

typeOf Expression Este tipo de expressão retorna o tipo associado a uma expressão ou variável.

Checked/Unchecked Expression Indica se um possível *overflow* aritmético deve ser verificado ou não.

Foreach Statement Simplificação do comando `for`.

Using Statement Define um escopo explícito para variáveis. Após este escopo, a variável não vale mais.

Jump Statement Comando `goto`. Pelo fato de C# definir explicitamente o comando `goto`, os comandos `break` e `continue` não podem ser rotulados como em Java.

Checked/Unchecked Statement Define um escopo em que o *overflow* de operações aritméticas é verificado(`checked`) ou é ignorado, mesmo que aconteça (`unchecked`).

À exceção das construções supracitadas, Java e C# possuem um vasto conjunto de similaridades sintáticas. Diferindo quase sempre apenas pelo uso de diferentes palavras reservadas. Sendo assim, achamos por bem construir o C#TS como sendo uma extensão de JaTS, reusando a maioria das construções. Em termos de representação dos nós sintáticos, não houve muita mudança. Exceto uma ou outra condição que foi relaxada, para permitir o reuso, por exemplo, a restrição em relação aos modificadores que podem ser adicionados a uma lista de modificadores.

A Figura 6.1 ilustra a relação entre os as liguagens Java e C#.

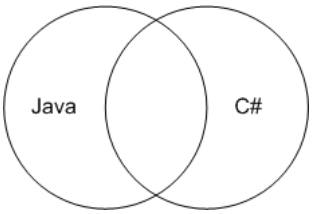

Figura 6.1 Relação entre as linguagens Java e C#

Como C#TS foi criado a partir da versão anterior de JaTS, para cada novo tipo de declaração, três classes foram geradas. Assim, para representar os conceitos novos de C#, foram criadas 23 classes para os nós sintáticos, mais os três *visitors*. Todas as classes foram geradas a partir de transformações JaTS.

Como a maioria dos nós da árvore foram reusados, optamos por implementar os *visitors* de C#TS como sendo extensões dos *visitors* de JaTS, com as definições somente das novas construções. Exceto o *visitor* de impressão, que precisou redefinir também outros métodos, para o tratamento de construções cuja sintaxe abstrata é a mesma, porém a sintaxe concreta é diferente (por exemplo, a construção *InstanceofExpression*. Em Java, usamos a palavra "`instanceof`" enquanto em C# usamos "`is`").

A Figura 6.2 ilustra a relação entre os *Visitors* de JaTS e C#TS.

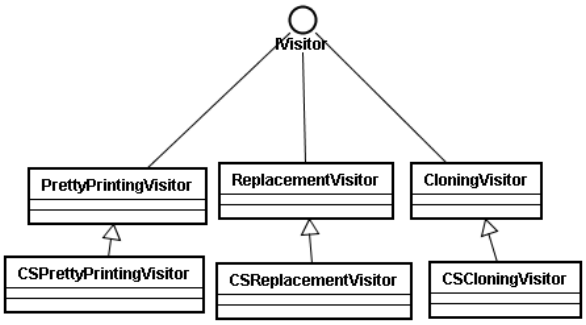

Figura 6.2 Relação entre os Visitors de JaTS e de C#TS

No que concerne à criação de uma linguagem de *templates* para para C#, C#TS-TL, os passos foram os mesmos mencionados na Seção 4.2.1 do Capítulo 4. E, embora tenha sido feito manualmente, não foi encontrado dificuldade, visto que muitas partes da definição de JaTS-TL foram reusadas.

6.3 UM SISTEMA DE TRANSFORMAÇÃO PARA OO1

O terceiro teste foi realizado com a linguagem OO1 [CIn06], uma linguagem desenvolvida em âmbito acadêmico apenas para ilustrar conceitos de orientação a objetos, que possui sintaxe bastante diferente da de Java e cuja implementação da árvore sintática era bastante diferente daquela que esperávamos, ou seja, com uma superclasse comum a todos os nós.

Neste, o esforço de modificação da árvore foi grande, pois alguns *refactorings* [FBB+99] tiveram de ser realizados antes de se aplicar as transformações. A árvore não estava definida de forma a facilitar a definição do método de casamento dentro do corpo de uma classe. Assim, tivemos de adicionar novos nós para tratar isto. Além disto, achamos por bem usar uma superclasse comum a todos os nós, para concentrar a implementação dos métodos da interface que não dependiam diretamente da classe de nó específica. Por exemplo, os métodos `isOptional`, `isVariable` e `isExecutable` podem ficar dentro desta superclasse comum. Diminuindo a quantidade de código repetido.

Após os refactorings, o código necessário para o sistema de transformação foi 80% gerado automaticamente. Diferente dos testes anteriores, aqui, algumas classes tiveram de ser desenvolvidas, a fim de que as transformações pudessem ser utilizadas, por exemplo, a classe que armazenava declarações de métodos.

Na Listagem 6.2 descrevemos um template de casamento em OOTS.

Listing 6.2 Template de casamento em OOTS

```
{
  classe #NOME_CLASSE {
    FieldDeclarationSet : #FDS;
    ProcDeclarationSet: #PDS
  }
  ;
  skip
}
```

A Listagem 6.3 descreve o *template* de geração em OOTS que adiciona o procedimento `renderJuros()` a uma classe em OO1.

Listing 6.3 Template de geração em OOTS

```
{
  classe #NOME_CLASSE {
    int juros = 10,
    FieldDeclarationSet : #FDS;
    proc renderJuros() {
      this.saldo := this.saldo + this.juros;
    },
    ProcDeclarationSet: #PDS
  }
  ;
  skip
}
```

Para ilustrar o resultado da transformação, imaginemos que o a transformação acima aplicada ao programa OO1 da Listagem 6.4.

Listing 6.4 Exemplo de programa em OO1

```
{
  classe Conta {
    int saldo = 0,
    string numero = "";
    proc printSaldo() {
      write("O saldo atual e':");
      write(this.saldo)
    }
  }
  ;
  skip
}
```

O resultado da aplicação seria a classe previamente definida, acrescida do atributo `juros` e do método `renderJuros()`, como mostra a Listagem 6.5:

Listing 6.5 Programa OO1 transformado

```
{
  classe Conta {
    int juros = 10, int saldo = 0,
    string numero = "";
    proc renderJuros() {
      this.saldo := this.saldo + this.juros;
    },
    proc printSaldo() {
      write("O saldo atual e':");
      write(this.saldo)
    }
 }
 ;
 skip
}
```

6.4 UM SISTEMA DE TRANSFORMAÇÃO PARA A LINGUAGEM DO JAVACC

Conforme vimos no Capítulo 5, o JavaCCTS é um sistema de transformação para a linguagem de especificação de gramáticas do JavaCC, criado com a abordagem sugerida neste trabalho.

A linguagem do JavaCC é um superconjunto sintático da linguagem Java. Assim, o JavaCCTS foi implementado de forma a reusar toda a estrutura de JaTS. A Figura 6.3 ilustra a relação entre os sistemas JaTS e JavaCC:

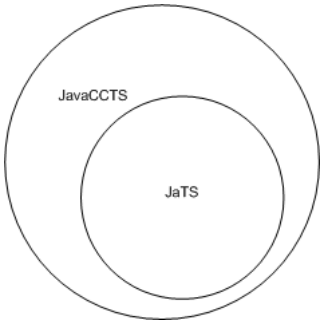

Figura 6.3 Relação entre JavaCCTS e JaTS

Como toda a implementação de JaTS foi reusada, a geração de código só ocorreu para as classes que representavam nós do JavaCC e os *visitors* foram definidos como extensões dos *visitors* presentes em JaTS. Além disto, como parte da árvore está sendo armazenada apenas como *string*, a escrita do *prettyprinting* foi facilitada.

No total, foram **20** classes adicionadas. Representando **8** produções da gramática que são suportadas. O *parser* foi modificado manualmente, mas isto não representou um problema, pois poucas produções foram modificadas. A parte referente à linguagem Java, usou os *parsers* presentes em JaTS. A Figura 6.4 mostra como os módulos de *IO* de JaTS e de JavaCCTS se relacionam.

Os números para a geração do JavaCC foram muito bons, dado que pouca coisa teve de ser implementada para que o sistema pudesse funcionar corretamente. Foram criados pouquíssimos métodos a mais para manipulação da API de JavaCCTS.

Métodos para criação de novas produções foram criados na classe `AbstractProduction`:

Listing 6.6 Classe AbstractProduction

```
public abstract class AbstractProduction {

    public abstract Production toExecutableProduction();
```

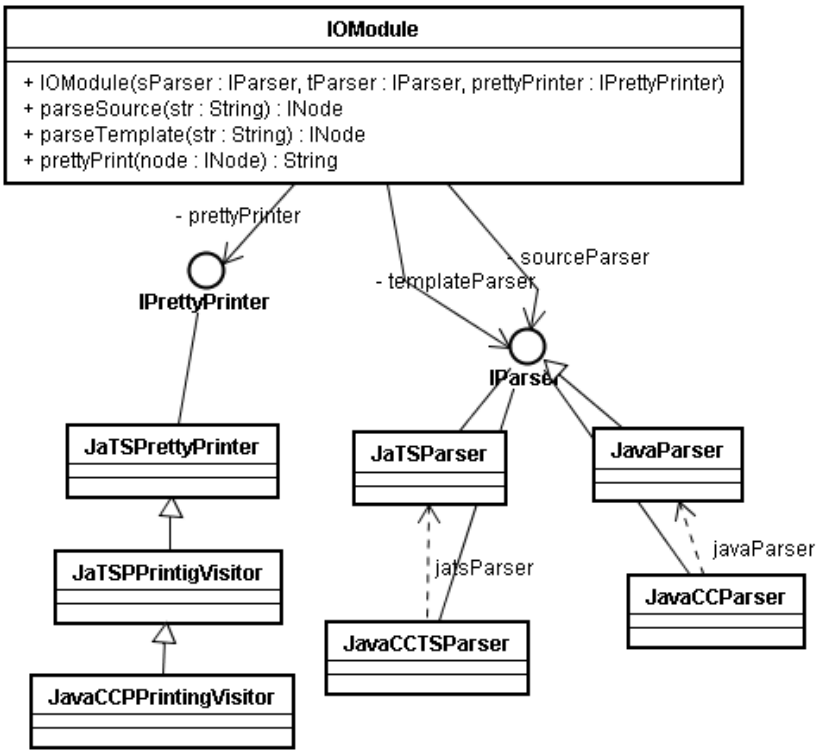

Figura 6.4 Relação entre os módulos de IO de JavaCCTS e JaTS

```
    public abstract Token toVariableName();

    public abstract Production toVariableProduction();
```

Método para a criação da produção para Expressões executáveis, o qual recebe os delimitadores da expressão executável e o nome da produção que é encapsulada pelos delimitadores:

Listing 6.7 Classe BnfProduction

```
public class BnfProduction extends Production{

   public static Production createExecExpProduction(String left,
                            String right,String production);
}
```

Método para criação de *tokens*. E um método específico para criação do *token* que indica uma variável.

Listing 6.8 Classe Token

```
public class Token {
   public static Token createToken(String tokenName){..}

   public static Token createToken(String tokenName, tokenValue){..}

   public static Token createVariableId(String varId,
                       String token, char variableChar){..}
}
```

6.5 RESUMO

Os exemplos testados atestam a boa qualidade da geração do código usando esta abordagem. Como o JavaCCTS foi o último dos sistemas a serem gerados, não pudemos aproveitar as suas facilidades para a alteração de gramáticas para geração de sistemas propostos neste trabalho.

Na Tabela 6.1 , mostramos em números, um resumo dos resultados da geração de

código. Estes números são calculados dividindo-se o *LOC* (número de linhas de código) do sistema logo após a aplicação das transformações de geração de código pelo *LOC* do sistema pronto. Como é a geração que introduz os métodos necessários ao sistema de transformação, queremos mostrar o quanto a quantidade de código gerada representa no sistema de transformação final. Se considerarmos o *LOC* inicial, este percentual praticamente não se altera, já que as linhas de código são mantidas na solução final. O número de linhas de código foi coletado usando a ferramenta JavaNCSS [Lee06].

Apesar de não terem sido obtidos através de experimentos formais [TGA03], estes números refletem o potencial de JaTS para gerar novos sistemas de transformação. Com a automação parcial da transformação da *linguagem objeto*, podemos obter resultados ainda melhores.

Tabela 6.1 Resultado da Geração de Novos Sistemas de Transformação

Linguagem	Esforço de Adaptação do Parser	Geração de código AST	Geração de Código Visitors	Reuso de Componentes
Java	baixo	92.44%	97.04%	total
C#	médio	89.61%	95.27%	total
OO1	médio	80.17%	97.25%	com alterações
JavaCC	baixo	90.00%	98.01%	total

O próximo passo na avaliação, seria usar o JavaCCTS para a transformação da gramática, fazendo com que a quantidade de código gerada seja maior e também, gerar sistemas de transformação para outros paradigmas, que não o orientado a objetos.

CAPÍTULO 7

CONCLUSÕES

Neste trabalho mostramos uma abordagem para a geração de uma linha de produtos de sistemas de transformação baseados em um conjunto de construções de metaprogramação introduzidos por JaTS. O custo de geração de um novo sistema, utilizando-se as transformações já implementadas, é baixo quando se compara aos benefícios de se ter sistema de transformação que leva em consideração a semântica da *linguagem objeto* e, conseqüentemente, pode exprimir transformações elaboradas de forma mais simples e concisa.

Os resultados obtidos na etapa de transformação da árvore sintática e na geração dos *visitors*, mostram que esta solução é adequada para a criação de novos sistemas de transformação.

Nos testes realizados durante o trabalho, a definição da linguagem de *templates* a partir da *linguagem objeto* foi feita manualmente. Mesmo assim, o custo da adaptação destas linguagens não foi alto, devido a particularidades das linguagens testadas. De forma que o ganho obtido na etapa de transformação da árvore e de geração dos *visitors* aliado ao reuso de componentes de apoio foi realmente representativo.

A definição do mecanismo para transformação de gramáticas definidas usando a ferramenta JavaCC, permitirá reduzir sensivelmente o trabalho manual, garantindo resultados ainda mais interessantes para esta abordagem.

Em resumo, as principais contribuições deste trabalho foram:

- Melhorias na estrutura de JaTS, permitindo maior reuso de seus componentes;
- Identificação dos componentes de um sistema de transformação baseado em *templates* e definição de uma abordagem para geração de uma linha de produtos de sistema de transformação baseados no JaTS;
- Definição de um sistema para transformação de especificações de gramática usando o JavaCC;
- Definição de três sistemas de transformação através da abordagem apresentada.

7.1 TRABALHOS RELACIONADOS

Existem algumas outras abordagens para permitir que sistemas de transformação possam transformar diferentes linguagens de programação. Em geral, estas ferramentas são dotadas de mecanismos que permitem uma certa independência de *linguagem objeto*. Para conseguir isso, este mecanismos, via de regra, perdem expressividade e só permitem regras de reescrita simples. Em nossa abordagem, pretendemos não perder o poder de expressividade da linguagem de *templates* e, por isso, pretendemos criar um sistema de transformação para cada linguagem que pretendemos usar como *linguagem objeto*.

7.1.1 Stratego

Stratego [Vis01b] é uma linguagem para especificação de sistemas de transformação baseados em estratégias de reescrita [Vis00].

Uma transformação em Stratego é definida em termos de assinatura, conjunto de regras e conjunto de estratégias. A árvore sintática abstrata (AST) do programa sendo transformado é representada através de um formalismo chamado SDF (Syntax Definition Formalism) [Vis97]. A assinatura declara os construtores destes termos. Regras condicionadas, são regras na forma: `L: l→ r where s`. Onde `L` define um nome para a regra, `l` define o padrão de casamento e `r` o padrão de saída. `s` define pré-condições sobre `l` que devem ser atendidas para que a regra seja aplicada. A estratégia define como as regras devem ser aplicadas e em que ordem.

A Listagem 7.1 mostra um exemplo de transformação em Stratego para transformar expressões lambda, usando redução beta.

Listing 7.1 Exemplo de transformação em Stratego

```
module lambda-transform

imports lambda-sig lambda-vars iteration simple-traversal

rules
  Beta : App(Abs(x, e1), e2) -> <lsubs>([(x,e2)], e1)

strategies
  simplify = bottomup(try(Beta))
  eager = rec eval(try(App(eval, eval)); try(Beta; eval))
  whnf = rec eval(try(App(eval, id)); try(Beta; eval))
```

Stratego traz a vantagem de usar um formalismo bastante poderoso para expressar as novas linguagens e tem sido usado em diversas aplicações. Uma especialmente útil, permite embutir linguagens de domínio específico em linguagens de propósito geral [VB04].

Em Stratego, as regras de reescrita são escritas usando-se a sintaxe concreta da *linguagem objeto*, entretanto, as regras de transformação são simples. Talvez, por esta razão, não foram encontradas na literatura revisada, implementação de reestruturações complexas usando Stratego.

7.1.2 TXL

TXL [Cor04, JCS02] é um sistema de transformação baseado em regras de reescrita, independente de *linguagem objeto*. Um metaprograma em TXL é composto pela descrição da gramática da *linguagem objeto* e por um conjunto de regras de transformação estruturais especificados como pares (padrão de casamento, regra de reescrita).

As regras de reescrita em TXL são homomórficas, ou seja, retornam sempre uma árvore do mesmo tipo da árvore recebida como parâmetro. Desta forma, é possível garantir que um programa TXL sempre gera saídas bem formadas de acordo com a sintaxe fornecida. Em JaTS, a garantia está no fato de que ele trata apenas uma linguagem e, portanto, conhece sua sintaxe. Por exemplo, uma declaração iterativa pode gerar uma declaração de atributo dentro do corpo de uma classe, mas não dentro do corpo de um método.

Exemplo de uma transformação usando TXL:

Listing 7.2 Exemplo de Transformação em TXL

```
% Inclui a gramática padrão de Pascal
include "Pascal.Grm"
% Refefine a gramática padrão, adicionando o novo statement
redefine statement
    ...
    | [reference] += [expression]

end redefine
% Transforma atribuições na forma "v += exp" em "v = v + exp "

rule main
    replace [statement]
        V [reference] += E [expression]
```

```
    by
        V := V + (E)
end rule
```

A linguagem de regras TXL é baseada num paradigma híbrido que combina aspectos de linguagens funcionais e de linguagens baseadas em regras lógicas. As estratégias de aplicação de regras não são explícitas como em *Stratego* [Vis01b] e o programador tem de introduzi-las diretamente nas regras de casamento e de decomposição de termos.

O processo de transformação consiste de fazer o *parsing* do programa de entrada, baseado na gramática provida e então sucessivamente aplicar as regras de transformação até que nenhuma transformação possa mais ser aplicada e então o código de saída é produzido.

O paradigma de programação de TXL é bastante diferente do paradigma de JaTS, exceto pelo fato de que ambos usam casamento de padrões e regras de substituição. JaTS permite construções poderosas para o casamento, como cláusulas opcionais e casamento semântico, o que torna a escrita de transformações mais concisa e expressiva. Além disso, expressões executáveis em JaTS embutem poder de processamento na linguagem sem perda de clareza e simplicidade.

Há um esforço no sentido de tornar o uso de TXL acessível para programadores ordinários através de uma família de linguagens de metaprogramação chamada μ* [CS92]. Uma característica é comum às linguagens μ*: os metaprogramas são descritos usando a *linguagem objeto* intercaladas com anotações em *Prolog* [CNV96], estas anotações são predicados sobre uma base de dados *Prolog* contendo fatos sobre a *linguagem objeto*. Apesar de transformações baseadas em *templates* serem amigáveis, o paradigma lógico continua sendo uma barreira para programadores ordinários.

A Listagem 7.3 mostra um metaprograma usando a notação μ* para a linguagem C. Ele gera o array `func` sempre que o predicado denotado pela cláusula `where` for satisfeito.

Listing 7.3 Metaprograma em C, usando a notação μ*

```
\  struct {
       char *name;
       int (*addr)();
    } func[] =
    {
      \$AllEntries,
      {"",0}
    };
```

```
\  where AllEntries
         \{$X,$Y} \[list init]
         each function (F [id])
         where X \"" \ [string] [" F]
         where Y \mpro \ [id]  [_ F]
```

7.1.3 DMS

DMS (Design Maintenance System) [BPM04], como TXL, é um sistema de transformação baseado em regras de reescrita com suporte a múltiplas linguagens. Atualmente suporta uma ampla gama de linguagens, desde linguagens de *scripting*, passando por linguagens estruturais até linguagens orientadas a objeto.

Uma transformação em DMS é composta por quatro elementos: o domínio (na notação de DMS, um domínio é um *parser* e um *prettyprinter* para uma dada linguagem) de entrada, o domínio de saída, as variáveis da transformação e a regra de reescrita, que é composta por um par padrão de entrada/padrão de saída, escritos, respectivamente, utilizando-se os domínios de entrada e de saída.

DMS pode ser usado tanto para transformações horizontais (no mesmo nível de abstração) quanto verticais (tradução). Vamos ilustrar estes dois cenários.

No primeiro, temos DMS sendo usado para reestruturação de programas. Neste caso, só é preciso definir um domínio (a linguagem C, neste exemplo). O símbolo `\v` indica uma variável. O trecho `(v: lvalue)` indica que só devem ser consideradas as variáveis que se encontrem no lado esquerdo de uma atribuição.

Listing 7.4 Exemplo de Reestruturação em DMS

```
default domain C;
   rule use-auto-increment(v: lvalue)
     :statement -> statement
     "\v = \v+1"
     rewrites to
     "\v++"
     if_no_side_effects(v);
```

Um exemplo de aplicação desta transformação:
Antes da transformação: `(*Z)[a>>2] = (*Z)[a>>2]+1;`

Depois da transformação: `(*Z)[a>>2]++;`

O outro cenário ilustra DMS sendo usado para uma transformação vertical. Neste exemplo, temos uma regra de transformação de código escrito na linguagem JOVIAL [Sch78](uma linguagem antiga, usada pelo departamento de defesa americano) para um programa escrito na linguagem C. Neste caso, precisamos dizer quem é o domínio de entrada (`default source domain`) e quem é o domínio de saída (`default target domain`).

Listing 7.5 Exemplo de Conversão de Programas em DMS

```
default source domain Jovial;
default target domain C;

private rule refine_data_reference_dereference_NAME
             (n1:identifier@C,n2:identifier@C)
             :data_reference->expression
    = "\n1\:NAME @ \n2\:NAME" -> "\n2->\n1".
private rule refine_for_loop_letter_2
             (lc:identifier@C,f1:expression@C,
             f2:expression@C,s:statement@C)
             :statement->statement
     = "FOR \lc\:loop_control :
        \f1\:formula BY \f2\:formula; \s\:statement"
       ->
        "{ int \lc = (\f1); for(;;\lc += (\f2)) { \s } }"
     if is_letter_identifier(lc).
```

Programa em Jovial:

Listing 7.6 Programa em Jovial

```
FOR i: j*3 BY 2 ;
    x@mydata = x@mydata+I;
```

Programa equivalente em C:

Listing 7.7 Programa em C

```
{ int i = j*3;
  for (;;i+=2){
```

```
            mydata->x = mydata->x + i
        }
    }
```

A forma geral de uma regra em DMS é a seguinte:

Listing 7.8 Regra em DMS

```
[modifier] rule_name "(" meta_var ("," meta_var)* ")" ":" type "->"
type "=" left_pattern "->" right_pattern ["if" condition]
```

Metavariáveis possuem um tipo que pode ser explicitamente pertencente a um domínio (por exemplo, `s:statement@C`):

```
meta_var = identifier ":" type ["@" identifier ]
```

Como vimos nos exemplos, o padrões `left_pattern` e `right_pattern` são descritos usando, respectivamente, o domínio de entrada e o domínio de saída. Dentro do padrão, uma variável é denotada usando o caracter "\". A condição pode ser descrita através de regras mais restritivas de casamento ou através de funções codificadas em PARLANSE [Bax06, BPM04], a linguagem de implementação de DMS.

As regras de reescrita em DMS permitem realizar vários tipos de transformação. Porém, regras mais elaboradas não podem ser construídas através da simples substituição de valores. Por exemplo, na geração de métodos de acesso, queremos que o nome do método seja gerado a partir de uma modificação no nome do atributo. Em JaTS, fazemos isso através de expressões executáveis (`#<#att.addPrefix("get")>#`). Não há registros na literatura revisada de suporte nativo a este tipo de construção (expressão executável). É possível, porém, estender o sistema através da definição de módulos usando a linguagem PARLANSE. Por ser definida usando um paradigma pouco usual, PARLANSE tem uma curva de aprendizado grande. Além disto, por ser uma linguagem proprietária, há pouca informação disponível a respeito desta linguagem.

7.1.4 MetaJ

Diferente dos outros sistemas apresentados aqui, MetaJ [Oli04, OBMB04] não é baseado em regras de reescrita. Ao invés disso, é uma ferramenta de metaprogramação baseado no paradigma orientado a objetos, que pode ser extensível para qualquer linguagem.

A única linguagem suportada nativamente é Java, mas é possível estendê-lo a partir da definição de *plug-ins* que tratam os aspectos diretamente dependentes de linguagem.

Um metaprograma em MetaJ é definido como uma classe Java. Sendo assim, MetaJ pode se integrar facilmente a qualquer programa Java, bastando, para tal, a importação dos seus módulos. O *framework* de MetaJ define operações comuns a várias linguagens e permite independência da linguagem sendo tratada. Para facilitar a descrição dos metaprogramas, MetaJ usa uma linguagem de *templates* cuja sintaxe é bastante similar a de JaTS-TL.

Assim como em nossa abordagem, esta linguagem de *templates* é gerada automaticamente por um dos componentes do sistema. Entretanto, a linguagem de *templates* proposta por MetaJ define poucas construções para os metaprogramas: apenas variáveis e cláusulas condicionais. Qualquer outro tipo de manipulação é feita mediante a definição de métodos em Java.

Assim, uma transformação de programas em MetaJ, é uma classe Java, contendo os métodos específicos para realizar a transformação. Métodos comuns a todas as linguagens são definidas no *framework* de MetaJ. Os *templates* são entidades que possuem nome e tipo e, antes de serem usados dentro dos metaprogramas, são compilados em classes Java. A Listagem 7.9 mostra um pequeno *template* descrito em MetaJ.

Listing 7.9 Exemplo de template em MetaJ

```
package myTemplates;
 language java // plug-in name
 template #compilation_unit MyTempl #{
    #[#packageDeclaration pck]#
    #[#import_declaration_list imps]#
    #[modifiers m ] class #identifier classId
              #super_opt [extends #name scName]# {
        #class_body_declarations cbds;
     }
}#
```

A Listagem 7.10 mostra a classe resultante da compilação do *template* da Listagem 7.9.

Listing 7.10 Código gerado a partir do template

```
package myTemplates;
import metaj.framework.AbstractTemplate;
public class MyTempl extends AbstractTemplate {
```

```
    public final Reference imps, td, m, classId, scName, cbds;
    Reference getClassId() {...}
    Reference getscName {...}
    ...
    //implementação dos métodos abstratos.

}
```

Todo *template* em MetaJ, quando compilado, gera pelo menos dois métodos na classe resultante: `boolean match(...)`, que serve para casar o padrão recebido como parâmetro com o padrão especificado pelo *template* e `String toString()`, que converte o *template* para a sua representação textual, substituindo todas as metavariáveis presentes pelos valores que lhes foram mapeadas anteriormente. Além disto, para cada meta-variável presente no *template*, é gerado um método de acesso correspondente.

As classes geradas podem, agora, serem usadas por um metaprograma:

Listing 7.11 Metaprograma

```
package myTemplates;
 import metaj.framework.AbstractTemplate;
 public class ChangeClassName {
   public static void main(String[] args){
      MyTempl tmpl = new MyTempl();
      // se a classe passada como parametro casar com
      // o template
      if (tmpl.matchFile(args[0])) {
        // verifica se o nome da classe é Conta
        // se for, muda para ContaAbstrata
        Preference className = temp.getClassId();
        if (className.equals("Conta")) {
           className.set("ContaAbstrata");
        }
      }
      // imprime a classe de saída num arquivo
      System.out.println(tmpl);
   }
 }
```

Esta integração direta com Java é interessante, porém, transformações mais complexas, exigem que muitos métodos sejam definidos, uma vez que a linguagem de *templates* é bastante simples. Transformações descritas de forma declarativa, como em JaTS, são mais simples de serem escritas pelos programadores, uma vez que a linguagem de *templates* é próxima da *template* da *linguagem objeto*.

7.2 TRABALHOS FUTUROS

Apesar dos bons resultados alcançados, acreditamos que há vários aspectos a serem explorados de modo a melhorar a forma com que estes sistemas são gerados e a qualidade dos sistemas finais. Aqui, listamos alguns dos trabalhos a que pretendemos dar continuidade. Esta lista, certamente, não é exaustiva e alguns outros trabalhos podem ser futuramente propostos:

Geração de *prettyprinter* Usando uma gramáticas anotada, é possível gerar um mecanismo para impressão do programa resultante. Este mecanismo pode estar integrado ao JavaCCTS.

Incorporar Tratamento de Contexto Extensão das novas funcionalidades de JaTS-TL para os demais sistemas, em especial o tratamento de contextos, definido por Santos [dS06].

Linguagem de Análise Geração da linguagem de análise para cada uma das linguagens apresentadas. As classes de análise podem ser reusadas, entretanto, o mecanismo de navegação, associada a construção do tipo `ExploreDeclaration` pode ser gerado através deta técnica.

Outros paradimas Este trabalho foi testado apenas para o paradigma orientado a objetos. Analisar sua aplicabilidade em outros paradigmas nos permitirá ver até que ponto o modelo empregado em JaTS pode ser considerado viável para sistemas de transformação em geral. Além disto, permitirá saber que pontos desta abordagem precisam ser modificados para que esta tenha um caráter mais generalista.

Integração Construir um ambiente integrado, com suporte a múltiplas linguagens. Os sistemas de transformação gerados por este trabalho compartilham várias classes. Um ambiente integrado permitiria ao programador realizar todas as transformações de que precisa sem mudança de contexto, o que pode aumentar sua produtividade. Possivelmente, estas transformações poderiam ser codificadas usando um mesmo metaprograma em JaTS-ML. Além disto, este tipo de integração, permitiria ao

ambiente, realizar certas transformações verticais. Por exemplo, já é possível usar JaTS e C#TS de forma integrada e realizar algumas transformações de Java para C# e vice-versa.

APÊNDICE A

TRANSFORMAÇÕES

A.1 TRANSFORMAÇÕES DE NÓS SINTÁTICOS

Listing A.1 Template de geração do nó sintático

```
package #PCKG.nodes;
import cin.jats.engine.util.TransformHelper;

import #PCKG.util.ResultSet;

import #PCKG.visitors.IVisitor;

import #PCKG.util.ResultSet;

ImportDeclarationSet:#NODE_IDS;


ModifierList:#NODE_MOD class #NODE #[ extends #NODE_SC ]#
                      #[implements NameList:#NODE_IFS ]# {

  FieldDeclarationSet:#FDS;
  InitializerSet:#INIT;

  public #NODE() {
    this.setTypeIdentifier(AbstractNode.#<#NODE.toConstantName()>#);
  }

  public  boolean equals( Object obj) {

    boolean result = false;
    if(obj instanceof #NODE) {
      #NODE #< #NODE.toVariableName() ># = (#NODE)obj;
      result = true;
```

```
      forall #fd #in #FDS {
        forall #vd #in #< #fd.getVariables() ># {
          let #varName = #< #NODE.toVariableName() >#;
              #vdId = #<vd.getIdentifier>#
              in
            #if(#fd.getType().isPrimitive()){
              result = result && this.#vdId == #varName.#vdId;
            }
            else {
              result = result &&
                 TransformHelper.equals(this.#vdId, #varName.#vdId);
            }
        }
      }
    }
    return result;
  }

  MethodDeclarationSet:#MDS;

  // Geração dos métodos de acesso apenas para atributos não
  // estáticos e não finais
  forall #A #in #FDS {

    #if(!#A.hasModifier("static")  !#A.hasModifier("final")) {
      forall #VD #in #< #A.getVariables() ># {
        let #name = #<#VD.getName()>#;
            #fType =#<#A.getType()>#;
          in
        // geração do Get ou Is
        #if(#A.getType().isBoolean()){
          public  #fType #< #VD.addPrefix("is") >#() {
            return this.#name;
          }
        }
        else {
          public  #fType #< #VD.addPrefix("get") >#() {
            return this.#name;
          }
```

```
      }
      // geração do set
      public  void #< #VD.addPrefix("set") >#(#fType #name) {
        this.#name = #name;
      }
    }
  }
}

public Object accept(IVisitor visitor, Object data)
  throws InconsistentNodeException, IllegalArgumentException,
  ExecutionException {
  if (!this.isExecutable() && !this.isVariable()) {
    forall #A #in #<#FDS.reverse()># {
      // só gera a chamada se o tipo for um descendente de INode
      #if (#A.getType() instanceof INode) {
        forall #vd #in #<#A.getVariables()># {
          if (this.#< #vd.getName() ># != null ){
            this.#< #vd.getName() >#.accept(visitor, data);
          }
        }
      }
    }
  }
  TransformHelper.invokeAcceptOnExecDec(visitor, data);
  return visitor.visit(this, data);
}

/**
 * Método que realiza o casamento entre dois nós
 */
public void match(INode node, ResultSet results)
     throws IllegalArgumentException, InconsistentNodeException,
            NodesNotMatchedException {

  ResultSet rs = (ResultSet) results.clone();
  if (!this.matchesAsAVariable(node, rs, INode.NO_TYPE)){
    let #varName = #<#TYPE.toVariableName()>#;
    in
```

```
        #TYPE  #varName = (#TYPE) node;
        forall #A #in #TYPE_ATTRS {
          let #attName = #<#A.getName()>#;
              #varType = #<#A.getType()>#;
          in
            #if (#varType instanceof INode) {
              this.#attName.match(#varName.#attName, rs);
            }
            else {
              #if (#varType.isPrimitive()){
                if (!(this.#attName == #varName.#attName)) {
                  throw new NodesNotMatchedException();
                }
              }
              else {
                if (!this.#attName.equals(#varName.#attName)) {
                  throw new NodesNotMatchedException();
                }
              }
            }
        }
    }
    results.merge(rs);
  }

  /**
   * Responsável por realizar o processamento no nó sintático
   */
  public Object process(Object data) throws ExecutionException,
    InconsistentNodeException, IllegalArgumentException {

    if (this.isExecutable()) {
      retorno = TransformHelper.processExecutableNode(this,data);
    }
    // Caso contrário, processa-se cada um dos filhos do nó.
    else {
      forall #A #in #FDS {
        #if(!(#A.getType().isSimple())){
          forall #vd #in #<#A.getVariables()># {
```

```
            let #tmpVar = #<#vd.addPrefix("tmp")>#;
                #getVar = #<#vd.addPrefix("get")>#;
                in
              #<#A.getType()># #var = this.#getVar();

              if (#tmpVar != null) {
                Object ret = this.processNode(#tmpVar, data);
                if (#tmpVar.isExecutable()) {
                  if (!(ret instanceof #<#A.getType()>#)){
                    throw new ExecutionException(this);
                  }
                  this.#<#vd.addPrefix("set")>#((#<#A.getType()># ) ret);
                }
              }
          }
        }
      }
    }
    return this;
  }
}
```

A.2 GERAÇÃO DOS VISITORS

A.2.1 Visitor de Substituição

Listing A.2 Template de geração do ReplacementVisitor

```
#[PackageDeclaration:#V_PCKG;]#

import cin.jats.engine.utl.ReplacementHelper;

ImportDeclarationSet:#V_IDS;

public class ReplacementVisitor extends AbstractVisitor {
```

```
    FieldDeclarationSet:#V_ATTRS;
    ConstructorDeclarationSet:#V_CDS;
    MethodDeclarationSet:#V_MDS;

    public Object visit(#TYPE node, Object data){
    if (node == null) {
      throw new IllegalArgumentException();
    }
    forall #A #in #FDS {
      let #tmpVar  = #<#A.addPrefix("tmp")>#;
          #varType = #<#A.getType()>#;
            in
          #varType #tmpVar = node.#<#A.addPrefix("get")>#();
          #if(#varType instanceOf INode) {
            if (#tmpVar.isVariable()) {
              node.#<#A.addPrefix("set")># (
                   (#<#A.getType()>#)ReplacementHelper.getValueOf(#tmpVar));
            }
          }
      return node;
    }
  }
}
```

A.2.2 Visitor de Clonagem

Listing A.3 Template de geração do CloningVisitor

```
#[PackageDeclaration:#V_PCKG;]#

ImportDeclarationSet:#V_IDS;

public class CloningVisitor extends AbstractVisitor {

  FieldDeclarationSet:#V_ATTRS;
  ConstructorDeclarationSet:#V_CDS;
```

```
  MethodDeclarationSet:#V_MDS;

  public Object visit(#NODE node, Object data)
      throws InconsistentNodeException, IllegalArgumentException {
    if (node == null) {
      throw new IllegalArgumentException("node is null");
    }
    #NODE resultNode = new #NODE();
    if (!node.isExecutable()  !node.isVarible()) {

      forall #A #in #NODE_ATTRS {
         #if(#A.getType() instanceOf INode) {
           forall #vd #in #<#A.getVariables()># {
             if(node.#<#vd.addPrefix("get")>#() != null &&
                  !(this.state.top() instanceof #<#A.getType()>#)){
               throw new InconsistentNodeException(node);
             }
             resultNode.#<#vd.addPrefix("set")>#(
                          (#<#A.getType()>#)this.state.pop());
           }
         }
      }

    }
    return resultNode;
  }
}
```

APÊNDICE B

GRAMÁTICA DO JAVACC

B.1 EXEMPLO DE TRANSFORMAÇÃO

B.1.1 Gramática de Entrada

Listing B.1 Gramática antes da transformação

```
options {
  JAVA_UNICODE_ESCAPE = true;
  STATIC = true;
}
PARSER_BEGIN(AritmeticGrammar) class AritmeticGrammar {
  public void test(){}
} PARSER_END(AritmeticGrammar)

SKIP : {
  " "
| "\t"
| "\n"
| "\r"
}

TOKEN : /* IDENTIFIERS */ {
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER: ["_","a"-"z","A"-"Z"] >
|
  < #DIGIT: ["0"-"9"] >
} TOKEN : /* LITERALS */ {
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
  >
|
```

```
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
}

SimpleNode Start() : {} {
  Expression() ";"
}


void Expression() : {} {
  AdditiveExpression()
}

void AdditiveExpression() : {} {
  UnaryExpression()
  ( ( "+" | "-" ) UnaryExpression() )*
}

void UnaryExpression() : {} {
  "(" Expression() ")" | Identifier() | Integer()
}

void Identifier() : {} {
  <IDENTIFIER>
}

void Integer() : {} {
  <INTEGER_LITERAL>
}
```

B.1.2 Gramática Resultante

Listing B.2 Gramática transformada

```
options {
  JAVA_UNICODE_ESCAPE = true;
  STATIC = true;
}
```

```
PARSER_BEGIN(AritmeticGrammar) class AritmeticGrammar {
  public void teste(){}
} PARSER_END(AritmeticGrammar)

SKIP : {
  " "
| "\t" | "\n" | "\r" }

TOKEN : /* IDENTIFIERS */ {
  < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
  < #LETTER: ["_","a"-"z","A"-"Z"] >
|
  < #DIGIT: ["0"-"9"] >
} TOKEN : /* LITERALS */ {
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
}

// gerado
TOKEN: {
  < VARIABLE_ID: "#"<IDENTIFIER> >
}

TOKEN: {
  <Integer: "Integer" >
  |
  <Start: "Start" >
  |
  <Expression: "Expression" >
  |
  <AdditiveExpression: "AdditiveExpression" >
  |
  <UnaryExpression: "UnaryExpression" >
}
```

```
SimpleNode Start() : {} {
  (LOOKAHEAD("Start" ":" VariableId())
  "Start" ":" VariableId()
  |
  LOOKAHEAD("Start" ":" ExecutableExpression())
  "Start" ":" VariableId()
  |
   Expression() ";"
  )
}


void Expression() : {} {

( LOOKAHEAD("Expression" ":" VariableId())
  "Expression" ":" VariableId()
  |
  LOOKAHEAD("Expression" ":" ExecutableExpression())
  "Expression" ":" VariableId()
  |
  AdditiveExpression()
 )
}

void AdditiveExpression() : {} {
  ( LOOKAHEAD("AdditiveExpression" ":" VariableId())
  "AdditiveExpression" ":" VariableId()
  |
  LOOKAHEAD("AdditiveExpression" ":" ExecutableExpression())
  "AdditiveExpression" ":" VariableId()
  |
  UnaryExpression() ( ( "+" | "-" ) UnaryExpression() )*
  )
}

void UnaryExpression() : {} {
 ( LOOKAHEAD("UnaryExpression" ":" VariableId())
  "UnaryExpression" ":" VariableId()
  |
```

```
  LOOKAHEAD("UnaryExpression" ":" ExecutableExpression())
  "UnaryExpression" ":" VariableId()
  |
  "(" Expression() ")" | Identifier() | Integer()
  )
}

void Identifier() : {} {  ( LOOKAHEAD("Identifier" ":" VariableId())
  "Identifier" ":" VariableId()
  |
  LOOKAHEAD("Identifier" ":" ExecutableExpression())
  "Identifier" ":" VariableId()
  |
  <IDENTIFIER>
  )
}

void Integer() : {} { ( LOOKAHEAD("Integer" ":" VariableId())
  "Integer" ":" VariableId()
  |
  LOOKAHEAD("Integer" ":" ExecutableExpression())
  "Integer" ":" VariableId()
  |
  <INTEGER_LITERAL>
  )
}
// producao nova
void VariableId(): {} {
   <VARIABALE_ID>
}
// producao nova
void ExecutableExpression() : {} {
   "#<" Expression() ">#"
}
```

B.2 GRAMÁTICA ORIGINAL DO JAVACC

B.2.1 Unidade de Compilação

Listing B.3 Unidade de compilação

```
javacc_input ::= javacc_options
  "PARSER_BEGIN" "(" <IDENTIFIER> ")"
  java_compilation_unit
  "PARSER_END" "(" <IDENTIFIER> ")"
  ( production )*
  <EOF>
```

B.2.2 Opções

Listing B.4 Opções de geração

```
javacc_options ::= [ "options" "{" ( option_binding )* "}" ]
option_binding ::= "LOOKAHEAD" "=" java_integer_literal ";"
  "CHOICE_AMBIGUITY_CHECK" "=" java_integer_literal ";"
  "OTHER_AMBIGUITY_CHECK" "=" java_integer_literal ";"
  "STATIC" "=" java_boolean_literal ";"
  "DEBUG_PARSER" "=" java_boolean_literal ";"
  "DEBUG_LOOKAHEAD" "=" java_boolean_literal ";"
  "DEBUG_TOKEN_MANAGER" "=" java_boolean_literal ";"
  "ERROR_REPORTING" "=" java_boolean_literal ";"
  "JAVA_UNICODE_ESCAPE" "=" java_boolean_literal ";"
  "UNICODE_INPUT" "=" java_boolean_literal ";"
  "IGNORE_CASE" "=" java_boolean_literal ";"
  "USER_TOKEN_MANAGER" "=" java_boolean_literal ";"
  "USER_CHAR_STREAM" "=" java_boolean_literal ";"
  "BUILD_PARSER" "=" java_boolean_literal ";"
  "BUILD_TOKEN_MANAGER" "=" java_boolean_literal ";"
  "TOKEN_MANAGER_USES_PARSER" "=" java_boolean_literal ";"
  "SANITY_CHECK" "=" java_boolean_literal ";"
  "FORCE_LA_CHECK" "=" java_boolean_literal ";"
  "COMMON_TOKEN_ACTION" "=" java_boolean_literal ";"
  "CACHE_TOKENS" "=" java_boolean_literal ";"
```

```
  "OUTPUT_DIRECTORY" "=" java_string_literal ";"
```

B.2.3 Produções

Listing B.5 Produções

```
production ::= javacode_production
  regular_expr_production
  bnf_production
  token_manager_decls


javacode_production ::= "JAVACODE"
  java_access_modifier java_return_type java_identifier "(" java_parameter_list
  java_block

bnf_production ::= java_access_modifier java_return_type
java_identifier "(" java_parameter_list ")" ":"
  java_block
  "{" expansion_choices "}"


regular_expr_production ::= [ lexical_state_list ]
  regexpr_kind [ "[" "IGNORE_CASE" "]" ] ":"
  "{" regexpr_spec ( "" regexpr_spec )* "}"

token_manager_decls ::= "TOKEN_MGR_DECLS" ":" java_block

lexical_state_list ::= "<" "*" ">"
  "<" java_identifier ( "," java_identifier )* ">"

regexpr_kind ::= "TOKEN"
  "SPECIAL_TOKEN"
  "SKIP"
  "MORE"
```

```
regexpr_spec ::= regular_expression [ java_block ] [ ":"
java_identifier ]

expansion_choices ::= expansion ( "" expansion )*

expansion ::= ( expansion_unit )*

expansion_unit ::= local_lookahead
  java_block
  "(" expansion_choices ")" [ "+"  "*"  "?" ]
  "[" expansion_choices "]"
  [ java_assignment_lhs "=" ] regular_expression
  [ java_assignment_lhs "=" ] java_identifier "(" java_expression_list ")"


regular_expression ::= java_string_literal
  "<" [ [ "#" ] java_identifier ":" ] complex_regular_expression_choices ">"
  "<" java_identifier ">"
  "<" "EOF" ">"
```

REFERÊNCIAS BIBLIOGRÁFICAS

[Bax06] Ira D. Baxter. *A PARallel LANguage for Symbolic Expression*. Semantics Design, http://www.semdesigns.com/Products/Parlanse/index.html, 2006.

[Bor06] Borland, http://www.borland.com/jbuilder. *JBuilder*, 2006.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society.

[Cas01a] Fernando Castor. Definição de uma Linguagem para Especificar Transformações em Java, 2001. Trabalho de Graduação.

[Cas01b] Fernando Castor. Uma semântica de ações para jats. Trabalho apresentado em disciplina., 2001.

[CB01] Fernando Castor and Paulo Borba. A language for specifying Java transformations. In *SBLP '01: Proceedings of the 5th Brazilian Symposium on Programming Languages*, pages 236–251, Curitiba, PR, BRAZIL, May 2001.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[CIn06] CIn/UFPE, http://www.cin.ufpe.br/∼in1007/linguagens/. *Linguagem Orientada a Objetos 1*, 2006.

[CNV96] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, 1996.

[Cor04] James R. Cordy. TXL - A language for programming language tools and applications. In *Proceedings of LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pages 1–27, 2004.

[COS+01] Fernando Castor, Kellen Oliveira, Adeline Souza, Gustavo Santos, and Paulo Borba. JaTS: A Java transformation system. In *XV Simpósio Brasileiro de Engenharia de Software*, pages 374–379, Outubro 2001.

[CS92] James R. Cordy and M. Shukla. Practical metaprogramming. In *Proceedings of the 1992 IBM Centre for Advanced Studies Conference*, pages 215–224, November 1992.

[dNS+02] Marcelo d'Amorim, Clóvis Nogueira, Gustavo Santos, Adeline Souza, and Paulo Borba. Integrating Code Generation and Refactoring. In *Workshop on Generative Programming, ECOOP02*, Malaga, Spain, June 2002. Springer Verlag.

[dS06] Gustavo Alexandre dos Santos. Suporte a refatorações em um sistema de transformação de propósito geral. Master's thesis, Universidade Federal de Pernambuco, Março 2006.

[Ecl06] Eclipse Foundation Inc., http://www.eclipse.org. *Eclipse*, 2006.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[GJSB96] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series. Addison-Wesley, 2th edition, 1996.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Java Series. Addison-Wesley, 3rd edition, 2005.

[GR04] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[Hal00] M. Hall. *Core Servlets and Java Server Pages*. Prentice Hall, 2000.

[HM01] E. Harold and W. Means. *XML in a Nutshell*. O'Reilly, 2001.

[Hud06] Scott E. Hudson. *Cup Parser Generator*. University of Princeton, http://www.cs.princeton.edu/ appel/modern/java/CUP/, 2006.

[JCS02] A.J. Malton J.R. Cordy, T.R. Dean and K.A. Schneider. Source transformation in software engineering using the txl transformation system. *Journal of Information and Software Technology*, 44:827–837, 2002.

[Jet06] JetBrains, http://www.jetbrains.com/idea/. *IntelliJ IDEA*, 2006.

[Lee06] Chr. Clemens Lee. *JavaNCSS: A Source Measurement Suite for Java*. http://www.kclee.de/clemens/java/javancss/, 2006.

[Lib03] Jesse Liberty. *Programming in C#*. O' Reilly, 2003.

[OBMB04] A. Oliveira, T. Braga, M. Maia, and R. Bigonha. Metaj: An extensible environment for metaprogramming in java. In *Proceedings of the VIII Brazilian Symposium on Programing Languages*, May 2004.

[Oli04] A Oliveira. MetaJ: Um ambiente para metaprogramação em java. Master's thesis, Universidade Federal de Minas Gerais, Março 2004.

[Opd92] Willian F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Applications Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pur06] Purdue University, USA, http://compilers.cs.ucla.edu/jtb/jtb-2003/index.html. *Java Tree Builder*, 2006.

[Qua06] Qualiti Software Processes, http://www.qualiti.com. *Qualiti Coder*, 2006.

[SB06] Gustavo Santos and Paulo Borba. Contextos de primeira classe em transformações de programas. In *Proceedings of the X Brazilian Symposium on Programing Languages*, pages 77–90, Itatiaia, RJ, Brazil, Maio 2006.

[Sch78] Jules I. Schwartz. The development of jovial. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, pages 203–214, New York, NY, USA, 1978. ACM Press.

[Sun06] Sun, https://javacc.dev.java.net/. *Java Compiler Compiler*, 2006.

[TGA03] G. Travassos, D. Gurov, and E. Amaral. Introdução a Engenharia de Software Experimental. Technical report, Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, Janeiro 2003.

[VB04] Eelco Visser and Martin Bravenboer. Concrete syntax for objects. domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM/SIGSOFT Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 365–383, October 2004.

[Vis97] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, Amsterdam, 1997.

[Vis00] Eelco Visser. *The Stratego Reference Manual*. Institute of Information and Computing Sciences, Utrecht University, 0.5 edition, 2000. Technical Documentation.

[Vis01a] E. Visser. A survey of rewriting strategies in program transformation systems, 2001.

[Vis01b] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. *Lecture Notes in Computer Science*, 2051:357+, 2001.

[Vis01c] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. system description of stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[W3C05] W3C, http://www.w3.org/MarkUp/. *Html Markup Language*, 2005.

[Wat91] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[WB99] David A. Watt and Derick Brown. JAS: A java action semantics. In P.D. Mosses and D.A. Watt, editors, *Second International Workshop on Action Semantics*, pages 43–53, http://www.brics.dk/Projects/AS/Workshop-99/BrownWatt.html, 1999. University of Aarhus, Denmark.

Este volume foi tipografado em LaTeX na classe UFPEThesis (www.cin.ufpe.br/~paguso/ufpethesis).